“GOOD ENOUGH” DATABASE CACHING

by

Hongfei Guo

A dissertation submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

2005

Abstract

Many application systems make use of various forms of asynchronously updated replicas to improve scalability, availability and performance. If an application uses replicas that are not in sync with source data, it is clearly willing to accept results that are not completely current, but typically with some limits on how stale the data can be. Today, such requirements are not explicitly declared anywhere; they can only be inferred from the properties of the replicas used. Because requirements are not explicit, the system cannot detect when they are not met and take appropriate action. Instead, data currency requirements are implicitly expressed in the application logic through the choice of data sources. This very much resembles the situation in the early days of database systems when programmers had to choose what indexes to use and how to join records.

This dissertation is about extending DBMS support for weaker consistency. We envision a scenario where applications are allowed to explicitly express relaxed currency and consistency (C&C) requirements in SQL; an administrator can explicitly express the desired local data C&C level in the cache schema; and query processing provides transactional guarantees for the C&C requirements of a query. Our research provides a comprehensive solution to this problem by addressing the following four issues: specifying “good enough” in SQL; building a constraint-based fine-grained “good enough” database caching model; enforcing “good enough” in query processing; and conducting a holistic system performance evaluation.

The first part of this dissertation proposes to allow queries to explicitly express relaxed C&C constraints. We extend SQL with a new currency clause, suggest a tentative syntax and develop a formal model that rigorously defines the semantics, thereby providing correctness standards for the use of replicated or cached data.

The second part of the dissertation develops a data quality-aware, fine-grained cache model and studies cache design in terms of four fundamental properties: *presence, consistency, completeness* and *currency*. Such a model provides an abstract view of the cache to the query processing layer, and opens the door for adaptive cache management.

The third part of this dissertation presents query processing methods for enforcing C&C constraints. We describe an implementation approach that builds on the MTCache framework for partially materialized views. The optimizer checks most consistency constraints and generates a dynamic plan that includes currency checks and inexpensive checks for dynamic consistency constraints that cannot be validated during plan compilation. Our solution not only supports transparent caching but also provides transactional fine grained data currency and consistency guarantees.

The last part of this dissertation reports a systematic performance evaluation. We establish a simplified but realistic model of a full-fledged database caching system in order to examine the influence of different workload assumptions, and different query processing and cache maintenance choices. This study reveals system characteristics under those assumptions and design choices, offering insights into performance tradeoffs.

Acknowledgements

Six Wisconsin winters have passed since I came to this department. Unlike many people, I don't mind the tough weather. I love this beautiful town and the people in the department, and will miss them both. The pursuit of a PhD degree has been challenging. It took sweat and sometimes even tears. Yet the feeling of reaching the goal is rewarding, and totally worth all the struggles. In retrospect, there have been many helping hands along my quest, without which I couldn't possibly have made it.

There is a Chinese proverb, "Once your teacher, always your father". I have two great advisors: Professor Raghu Ramakrishnan and Dr. Paul Larson. What I have learned from them is far beyond technical, and it will benefit me for years to come. Raghu taught me to always have higher standards for myself. No matter what you do, do your best. I remember often looking forward to meeting with him, because it would be enlightening; at the same time I was nervous, for the meeting was always intensive and required extensive homework. When I first came to him, I had only the determination to get the PhD, but not quite knowing how. With patience and trust, Raghu trained me into an able researcher.

Paul was my mentor the first time I interned in Microsoft Research, and has been my co-advisor ever since. Whenever I have technical questions or want to discuss some ideas, I can always rely on him for guidance. An expert in the SQL Server codebase, he provided me with hand-in-hand help in the implementation and experimental parts of the dissertation. Raghu being away in India, Paul took the responsibility to supervise the performance modeling part and the writing of this dissertation. He taught me how to do high quality system work and to always be down to earth.

It was bad timing for potential database students when I first arrived — too many students and only two faculty members. My enthusiasm for joining the best database group simply hit the wall of reality. Although already busy with existing students, Professor Jeff Naughton gave me hope by taking me for independent study. I still vividly recall how my frustration and confusion turned into excitement when I received his email. It was him who started my interest in caching and encouraged me to look into the direction of caching with relaxed data quality. Not only did he help me find funding (Microsoft Graduate Fellowship), but he also recommended me to Paul and Raghu. His door was always open for me.

I would also like to thank Professor David DeWitt for all his contributions to the department and for his support. It is from him that I learned an important attitude: don’t take things for granted with complex systems. Build the system, and let the facts talk. The performance modeling part of the dissertation was inspired by discussions with him. Thanks also go to Professors Stephen Wright and Yibin Pan for being on my dissertation committee.

Part of this research was done during two internships in the Database Group at Microsoft Research. I would like to thank Jonathan Goldstein, a good colleague and friend. He shared his experience as a graduate student with me and gave me valuable advice. Part of my financial support during graduate school was provided by a Microsoft Graduate Fellowship. Microsoft also generously sponsored several of my conference trips.

My work on performance modeling could not have been completed without the assistance of my colleague Krishna Pradeep Reddy Tamma. He helped me to implement the model and write scripts for producing the experiments. Pradeep went well above and beyond the call of duty in helping me accomplish that work.

I was very fortunate to work in a group of fantastic people. Discussions with and feedback from them benefited my research. Their kindness and support gave me courage and warmed my heart. Specifically, I would like to thank Ahmed Ayad, Jennifer Beckham, Pedro Bizarro, Bee-Chung Chen, Lei Chen, Vuk Ercegovac, Alan Halverson, Ameet Kini and Kristen Lefevre. My thanks also go to alumni Jianjun Chen, Leonidas Galanis, Rajasekar Krishnamurthy, Qiong Luo, Neoklis Polyzotis, Kristin Tufte, Efstratios Viglas and Chun Zhang. I would also like to thank Josef Bruger, who kindly took the time and effort to recover my (half a year's worth of) experimental data from a corrupted disk.

I would like to thank my fellow graduate students and good friends Hao Wang, Hong Lin, Shiliang Hu, Yunpeng Li and Min Xu. Their friendship made this journey enjoyable.

A big thank you to the administrative staff members in the department: Lorrene Webber, Brian Asen, Melody Bakken, Robert Holloway, John O'Malley, Cache Richard and Virginia Werner. Their kindness and support makes life in the department smooth.

If I were to name one person who deserves the most thanks, it would be my husband-to-be, Bruno Denuit. Our long distance relationship started 5 years ago and his love and support for me since has never faltered. He makes me smile even in gloomy days, and gives me the courage to fight on. I thank my future mother-in-law Betty Denuit-Cordenier and my brother, Jiannan Guo, for their love and encouragement. I thank my parents, Kaiwang Guo and Guangyun Tan. Thank you for raising me up, giving me all your love, a free space to develop my character, and planting in my heart the confidence to excel.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Many application systems today make use of various forms of asynchronously updated replicas to improve scalability, availability and performance. We use the term replica broadly to include any saved data derived from some underlying source tables, regardless of where and how the data is stored. This covers traditional replicated data and data cached by various caching mechanisms. "Asynchronously updated" simply means that the replica is not updated as part of a database transaction modifying its source tables; the state of the replica does not necessarily reflect the current state of the database.

If an application uses replicas that are not in sync with the source data, it is clearly willing to accept results that are not completely current, but typically with some limits on how stale the data can be. For instance, a typical e-commerce site such as eBay makes frequent use of replicated and cached data. When browsing auctions in a category, the data (e.g., item prices, number of bids) may be a little out of date. However, most users understand and accept this, as long as the page they see when they click on an individual auction is completely current. As a concrete example, consider the following query that returns a summary of books with the specified title:

```
SELECT  *
FROM    Books B, Reviews R
WHERE   B.isbn = R.isbn AND B.title = “Databases”
```

Different applications may have different freshness requirements for this query. Application A needs an up-to-date query result. Application B prefers a low response time but doesn’t care if the reviews are a bit stale. Application C does not mind if the result is stale but it requires the entire result to be snapshot consistent, i.e., reflect a state of the database at a certain point of time. Application D is satisfied with a weaker version of this guarantee, requiring only that all rows retrieved for a given book reflect the same snapshot, with different books possibly from different snapshots.

Application designers normally understand when it is acceptable to use copies and what levels of data staleness and inconsistency are within the application’s requirements. Currently, such requirements are only implicitly expressed through the choice of data sources for queries. For example, if a query Q1 does not require completely up-to-date data, we may design the application to submit it to a database server C that stores replicated data instead of submitting it to database server B that maintains the up-to-date state. Another query Q2 accesses the same tables but requires up-to-date data so the application submits it to database server B. The routing decisions are hardwired into the application and cannot be changed without changing the application.

Because requirements are not explicit, the system cannot detect when they are not met and take appropriate action. For example, the system could return a warning to the application or use another data source.

This very much resembles the situation in the early days of database systems when programmers had to choose what indexes to use and how to join records. This was remedied by raising the level of abstraction, expressing queries in SQL and making the database system responsible for finding the best way to evaluate a query. We believe the time has come to raise the level of abstraction for the use of replicated and cached data by allowing applications to state their data currency and consistency requirements explicitly and having the system take responsibility for producing results that meet those requirements.

This dissertation focuses on extending DBMS support for relaxed consistency. We envision a scenario where applications are allowed to explicitly express relaxed currency and consistency (C&C) requirements in SQL; an administrator can explicitly express the desired local data C&C level in the cache schema; query processing provides transactional guarantees for the C&C constraints of a query; and the cache makes intelligent decisions on scarce resource allocation. Our research provides a comprehensive solution to this problem by addressing the following four issues: specifying “good enough” in SQL; building a constraint-based fine-grained “good enough” database caching model; enforcing “good enough” in query processing; and conducting a holistic system performance evaluation.

The first part of this dissertation [GLRG04] proposes to allow queries to explicitly express relaxed C&C constraints. We extend SQL with a new currency clause and suggest a tentative syntax. We also develop a formal model that rigorously defines the semantics, thereby providing correctness standards for the use of replicated or cached data.

The second part of the dissertation [GLR05a, GLR05b] provides a fine-grained data quality-aware cache model. We build a solid foundation for cache description by formally defining four fundamental properties of cached data: presence, consistency, completeness and currency. We introduce a novel cache model that supports a specific way of partitioning cached data, and translate a rich class of integrity constraints (expressed in extended SQL DDL syntax) into properties required to hold over different partitions. We identify an important property of cached views, called safety, and show how safety aids in efficient cache maintenance. Further, we formally define cache schemas and characterize when they are safe, offering guidelines for cache schema design.

The third part of this dissertation [GLRG04, LGGZ04] develops query processing methods for enforcing C&C constraints. First, for a simple case where each view in the cache is consistent and complete, we implement a prototype in MTCache, our mid-tier database cache built on the Microsoft SQL Server codebase. The optimizer checks the consistency constraints during plan compilation and generates a dynamic plan that includes currency checks. A SwitchUnion plan operator checks the currency of each local replica before use and switches between local and remote sub-plans accordingly. We integrate the C&C constraints of a query and replica update policies into the cost-based query optimizer. This approach supports transparent caching, and makes optimal use of the cache DBMS, while guaranteeing that applications always get data with sufficient quality for their purpose.

We then extend the framework for a more general case where we keep track of cache properties at the granularity of partitions of a view [GLR05a, GLR05b]. The optimizer checks most consistency constraints during plan compilation and generates a dynamic plan that includes currency checks and inexpensive checks for dynamic consistency constraints that cannot be validated during plan compilation.

The last part of this dissertation reports a systematic performance evaluation. We begin by establishing a performance evaluation framework based on a model of a database management system. Our model captures the main elements of a database environment, including both *users* (i.e., terminals, the sources of transactions) and *physical resources* for storing and processing the data (i.e., disks and CPUs), in addition to the characteristics of the workload and the database. Then we extend the single-site model to a cache-master configuration, capturing the interaction between the cache and the master. In addition, we refine the single-site model to reflect the characteristics of cache organization and maintenance.

Based on this framework, we examine the influence of different workload assumptions, and different query processing and cache maintenance choices. This study reveals system characteristics under those assumptions and design choices, offering insights into performance tradeoffs.

Although this dissertation focuses on a database caching environment, the philosophy of our solutions can be applied to a broad range of usage scenarios where the system can provide additional functionality if applications explicitly state their C&C requirements.

Traditional replicated databases: Consider a database containing replicated data propagated from another database using normal (asynchronous) replication. The system can easily keep track of how current the data is, but today that information is not exploited. If an application states its currency requirements, the system could detect and take action when the application's requirements are not met. Possible actions include logging the violation, returning the data but with an error code, or aborting the request.

Mid-tier database caching: This scenario was motivated by current work on transparent mid-tier database caching as described in [LGZ04, ABK+03, BAK+03]. Suppose we have a back-end database server that is overloaded. To reduce the query load, we replicate part of the database to other database servers that act as caches. When a cache DBMS receives a query, it attempts to answer it from the local data and if that is not possible it forwards the query transparently to the back-end server. In this scenario, it is crucial to know the C&C constraints so that the cache DBMS can decide whether local data can be used or not.

Caching of query results: Suppose we have a component that caches SQL query results (e.g., application level caching) so that those results can be reused if the same query is submitted later. The cache can easily keep track of the staleness of its cached results and if a result does not satisfy a query's currency requirements, transparently recompute it. In this way, an application can always be assured that its currency requirements are met.

The rest of the dissertation is organized as follows. Chapter 2 presents the SQL extension, which allows an individual query to specify fine-grained C&C requirements. We develop a data quality-centric database caching model in Chapter 3, providing flexible local data quality control for cache administration in terms of granularity and cache properties. In Chapter 4, we introduce the framework to enforce C&C constraints in MTCache, a prototype transparent mid-tier database cache built on Microsoft SQL Server codebase, for the simple case where each view in the cache is consistent. We remove this restriction, and generalize the algorithms to support fine-grained C&C checking in Chapter 5. Chapter 6 establishes a model of a full-fledged database caching system and reports performance evaluation results. We discuss related work in Chapter 7 and conclude in Chapter 8.

Chapter 2

Specifying Data Quality Constraints in SQL

Different applications might have different data quality requirements. We define a model for relaxed currency and consistency (C&C) constraints, allowing an individual query to express its fine-grained data quality requirements. Section 2.1 describes how to specify C&C constraints through a simple extension of SQL syntax. We start with constraints for read-only transactions; first for single-block queries, and then generalizing to multi-block queries. Then we introduce an additional constraint for read-write transactions. We build a formal model in Section 2.2, which not only defines the semantics of the set of C&C constraints specified by the proposed SQL extension, but also covers general C&C constraints. It thereby provides correctness standards for general use of replicated and cached data.

2.1 Specifying Currency and Consistency Constraints

In this section we introduce our model for currency and consistency constraints by means of examples. We propose expressing C&C constraints in SQL by a new currency clause and suggest a tentative syntax. The semantics of C&C constraints are described informally in this section; the following one contains formal definitions.

Before proceeding, we need to clarify what we mean by the terms currency and consistency. Suppose we have a database with two tables, Books and Reviews, as might be used by a small online book store.

Replicated data or the result of a query computed from replicated data may not be entirely up-to-date. Currency (staleness) simply refers to how current or up-to-date we can guarantee a set of rows (a table, a view or a query result) to be. Suppose that we have a replicated table BookCopy that is refreshed once every hour. In this scenario, the currency of BookCopy is simply the elapsed time since this copy became stale (i.e., when the first update was committed to Books after BookCopy’s last refresh) to the commit time of the latest update transaction on the back-end database.

Suppose we have another replicated table, ReviewCopy, which is also refreshed once every hour. The state of BookCopy corresponds to some snapshot of the underlying database and similarly for ReviewCopy. However, the states of the two replicas do not necessarily correspond to the same snapshot. If they do, we say that they are mutually consistent or that they belong to the same consistency group. Whether or not the two replicas are mutually consistent depends entirely on how they are updated.

2.1.1 Single-Block Queries

To express C&C constraints we propose a new currency clause for SQL queries. The new clause occurs after a Select-From-Where (SFW) block and follows the same scoping rules as the WHERE clause. Specifically, the new clause can reference tables defined in the current

```
Q1:   SELECT    *
      FROM      Books B, Reviews R
      WHERE     B.isbn = R.isbn and B.title = “Databases“

E1: CURRENCY BOUND 10 min ON (B, R)
E2: CURRENCY BOUND 10 min ON (B), 30 min ON (R)
E3: CURRENCY BOUND  10 min ON (B) BY B.isbn,
                    30 min ON (R) BY R.isbn
E4: CURRENCY BOUND 10 min ON (B, R) BY B.isbn
```

Figure 2.1 Single-block example C&C constraints

or in outer SFW blocks. We use query Q1 to illustrate different forms of the currency clause and their semantics, as shown in Figure 2.1. The query is a join of Books and Reviews.

Currency clause E1 expresses two constraints: a) the inputs cannot be more than 10 min out of date and b) the the two input tables must be consistent, that is, from the same database snapshot. We say that B and R belong to the same consistency class.

Suppose that we have cached replicas of Books and Reviews, and compute the query from the replicas. To satisfy the C&C constraint, the result obtained using the replicas must be equivalent to the result that would be obtained if the query were computed against some mutually consistent snapshots of Books and Reviews, that are no older than 10 min (when execution of the query begins).

E2 relaxes the bound on R to 30 min and no longer requires that the tables be mutually consistent by placing them in different consistency classes. The easiest way to construct a currency clause is to first specify a bound for each input and then form consistency groups by deciding which inputs must be mutually consistent.

E1 and E2 require that every Books row be from the same snapshot and similarly for Reviews, which may be stricter than necessary. Sometimes it is acceptable if rows or groups of rows from the same table are from different snapshots. E3 and E4 illustrate how we can express different variants of this requirement.

We assume that isbn is a unique key of Books. E3 allows each row of the Books table to originate from different snapshots (because B.isbn is unique). The phrase "(R) by R.isbn" has the following meaning: if the rows in Reviews are grouped on isbn, rows within the same group must originate from the same snapshot. Note that a Books row and the Reviews rows it joins with may be from different snapshots (because Books and Reviews are in different consistency classes).

In contrast, E4 requires that each Books row be consistent with the Reviews rows that it joins with. However, different Books rows may be from different snapshots.

In summary, a C&C constraint in a query consists of a set of triples where each triple specifies

1) a currency bound,
2) a set of tables forming a consistency class, and
3) a set of columns defining how to group the rows of the consistency class into consistency groups.

The query-centric approach we have taken for dealing with asynchronously maintained copies is a fundamental departure from *maintenance-centric* prior work on replica management (see Chapter 7), which concentrates on maintenance policies for guaranteeing different kinds of constraints over cached objects. While this earlier work can be leveraged by a system in determining what constraints hold across a set of cached objects, the user's C&C requirements in the query ultimately determine what copies are acceptable, and the system must guarantee that these requirements are met, if necessary by fetching master versions of objects. This is the focus of this thesis.

An important consequence of our approach is a significant difference in workloads, because C&C constraints influence when and how caches need to be updated, necessitating new cache management policies and mechanisms. However, this issue is beyond the scope of this thesis.

```
Q2:                                 Q3:

SELECT  *                           SELECT  *
FROM    Sales S,                    FROM    Books B, Reviews R
(                                   WHERE   B.isbn = R.isbn
 SELECT  *                              AND B.isbn IN
 FROM    Books B, Reviews R         (
 WHERE   B.isbn = R.isbn             SELECT  isbn
 CURRENCY BOUND                      FROM    Sales S
        5 min ON (B, R)              WHERE   year='2003'
) T                                  CURRENCY BOUND
WHERE   S.isbn = T.isbn                     10 min ON (S, B)
    AND year= '2003'                )
CURRENCY BOUND                      CURRENCY BOUND
        10 min ON (S, T)                    10 min ON (B, R)
```

Figure 2.2 Multi-block example C&C Constraints

2.1.2 Multi-Block Queries

An SQL query may, of course, consist of multiple SFW blocks. C&C constraints are not restricted to the outermost block of a query — any SFW block can have a C&C constraint. If a query contains multiple constraints, all constraints must be satisfied.

We first consider subqueries in the FROM clause. Suppose we have an additional table Sales with one row for each book sale and consider the query Q2 in Figure 2.2. Note that such queries can arise in a variety of ways. For instance, the original query may have referenced a view and the query in the FROM clause is the result of expanding the view.

Whatever input data the query is computed from, the inputs must be such that both constraints are satisfied. The outer currency clause states that S must be from the same snapshot as T. But T is computed from B and R, which implies that S, B and R must all be from the same snapshot. If they are from the same snapshot, they are all equally stale. Clearly, to satisfy both constraints, they must be no more than 5 min out of date. In summary, the least restrictive constraint that the inputs must satisfy is “5 min ON (S, B, R)”.

Next we consider subqueries in clauses other than the FROM clause. For such subqueries we must also decide whether the inputs defined in the subquery need to be consistent with any of the inputs in an outer block. We modify our join query Q1 by adding a subquery that selects only books with at least one sale during 2003, see Q3 in Figure 2.2.

When constructing the currency clause for the subquery, we must decide whether S (Sales) needs to be consistent with B and/or R (in the outer block). If S must be consistent with B, we simply add B to the consistency class of S, see Q3. Because the outer currency clause requires that R be consistent with B, it follows that B, R, and S must all be consistent, that is, they all form a single consistency class.

If S need not be consistent with any tables in the outer block, we simply omit the reference to B and change the inner currency clause to “10 min on S”.

2.1.3 Timeline Consistency

Until now we have considered each query in isolation. Given a sequence of queries in a session, what constraints on the relationships between inputs to different queries are of interest? Even though not explicitly stated, current database systems provide an important guarantee on sequences of queries within the same session: time moves forward. If a user reads a row R twice but row R is updated and the change committed in between the reads, then the second read will see the updated version of R.

This rather natural behavior follows from the fact that queries use the latest committed database state. However, if queries are allowed to use out-of-date replicas and have different currency bounds, there is no automatic guarantee that perceived time moves forward. Suppose queries Q_1 and then Q_2 are executed against replicas S_1 and S_2, respectively. S_2 is not automatically more current than or equal to S_1; the ordering has to be explicitly enforced.

We take the approach that forward movement of time is not enforced by default and has to be explicitly specified by bracketing the query sequence with “**begin timeordered**” and “**end timeordered**”. This guarantees that later queries use data that is at least as fresh as the data used by queries earlier in the sequence.

This feature is most useful when two or more of the queries in a sequence have overlapping input data. In this case, we may get very counterintuitive results if a later query were to use older data than an earlier one. Note that users may not even see their own changes unless timeline consistency is specified, because a later query may use a replica that has not yet been updated.

2.2 Formal Semantics of Currency and Consistency Constraints

In this section, we build a formal model that defines the semantics of general currency and consistency constraints. We classify currency and consistency requirements into four types: *per-object*, *per-group*, *inter-group,* and *inter-statement*. Our approach to currency and consistency constraints in a query specification reflects two principles:

1) C&C constraints of query results should not depend on data objects not used in constructing the result; this is achieved through the use of the extended query (Section 2.2.2).
2) It must be possible to require consistency for any subsets of the data used in a query; we achieve this, naturally, by leveraging the query mechanism to identify the subsets (Section 2.2.3.3).

2.2.1 A Model of Databases with Copies

A *database* is modeled as a collection of *database objects* organized into one or more tables. Conceptually, the granularity of an object may be a view, a table, a column, a row or even a single cell in a row. To be specific, in this thesis an object is a row. Let identity of objects in a table be established by a (possibly composite) key K. When we talk about a key at the database level, we implicitly include the scope of that key. Every object has a *master* and zero or more *copies*. The collection of all master objects is called the **master database**. We denote the database state after n committed update transactions ($T_1..T_n$) by $H_n = (T_n \circ T_{n-1} \circ \ldots \circ T_1(H_0))$, where H_0 is the initial database state, and "$\circ$" is the usual notation for functional composition. Each database state H_i is called a **snapshot** of the database. Assuming each committed transaction is assigned a unique timestamp, we sometimes use T_n and H_n interchangeably.

A *cache* is a collection of (local) materialized views, each consisting of a collection of copies (of row-level objects). Although an object can have at most one copy in any given view, multiple copies of the same object may co-exist in different cached views. We only consider local materialized views defined by selection queries that select a subset of data from a table or a view of the master database.

Transactions only modify the master database, and we assume Strict 2PL is enforced. Further, for simplicity we assume that writers only read from the master database. Copies of modified objects are synchronized with the master by the DBMS after the writer commits through (system-initiated) *copy-transactions,* but not necessarily in an atomic action as part of the commit.

Next, we extend the database model to allow for the specification of currency and consistency constraints. We emphasize that the extensions described below are conceptual; how a DBMS supports these is a separate issue.

Self-Identification: **master**() applied to an object (master or copy) returns the master version of that object.

Transaction Timestamps: The function **xtime**(T) returns the transaction timestamp of transaction T. We overload the function xtime to apply to objects. The transaction timestamp associated with a master object O, **xtime**(O, H_n), is equal to xtime(A), where A is the latest transaction in $T_1..T_n$ that modified O. For a copy C, the transaction timestamp **xtime**(C, H_n) is copied from the master object when the copy is synchronized.

Copy Staleness: Given a database snapshot H_n, a copy C is stale if master(C) was modified in H_n after xtime(C, H_n). The time at which O becomes stale, called the *stale point*, **stale**(C, H_n), is equal to xtime(A), where A is the first transaction in $T_1..T_n$ that modifies master(C) after xtime(C, H_n). The **currency** of C in H_n is measured by how long it has been stale, i.e., **currency**(C, H_n) = xtime(T_n) – stale(C, H_n).

A read-only transaction's read requests include *currency and consistency constraints*, and any copy of the requested object that satisfies the constraints can be returned to the transaction. We assume that as transactions commit, the DBMS assigns them an integer id — a timestamp — in increasing order.

2.2.2 The Extended Query Set

Intuitively, the C&C requirements of query results should not depend on data objects not used in constructing the result. For a given query Q, we construct an **extended query set** $\mathbf{Q^{ext}}$, which consists of a set of extended versions of Q, termed **extended queries**, denoted by

Q_i^{ext}. For a simple query (single block, without referring to any views), $\mathbf{Q^{ext}}$ contains only one query. For more general queries, the construction proceeds block-at-a-time, starting from the outmost one, and finding one query for each sub-block or view, ensuring that the result of $\mathbf{Q^{ext}}$ includes all objects used in constructing the result of Q (including objects used in WHERE clauses, grouping, etc.). We refer to the result of $\mathbf{Q^{ext}}$, a set of derived tables, as the **relevant set** for Q. We first describe the construction for simple queries and then for generalized cases.

2.2.2.1 Simple queries

For a simple query, i.e., a single-block query that does not refer to any views, the extended query is the original query without aggregates, but with the projected out attributes that are referred to in the query. Given a query Q in the following format:

```
SELECT  <select-list>     FROM   <from-list>
WHERE   <where-constraints>
[GROUP BY <group-by-key>]
[HAVING <having-constraints>]
```

Where the from-list contains only base tables. The construction of the extended query Q^{ext} is described by algorithm ConstructExtendedQuery shown in Figure 2.3.

2.2.2.2 Queries with Nested Queries

Consider a query Q with n levels of nested queries, numbered from outmost to innermost, of which nested query Q_i, i = 1..n, with Q_1 = Q, is defined recursively as follows:

```
SELECT  <select-list_i>
FROM    <from-list_i>
WHERE   <where-constraints_i> AND (OR)
        p_i( $attr_{i1}$, …, $attr_{im_i}$ , Q_{i+1})
[GROUP BY <group-by-key_i>]
[HAVING <having-constraints_i>]
```

Where p_i (i=1,…,n–1) is a predicate defined on a set of attributes from tables in the from-list and the result of nested query Q_{i+1}, and p_n is *true*. We proceed from outmost to innermost, block-by-block, to find the extended query for each nested query. For each nested query Q_i, i=1..n, we first apply ConstructExtendedQuery. If Q_i is not correlated, we are done. Otherwise, we only want to keep in the result of the extended query rows that are correlated to its outer blocks. This is achieved by replacing any attribute A that are from an outer block Q_j (j<i) by Q_j^{ext}.A and adding Q_j^{ext} to the from-list. Note that for simplicity, we use Q_j^{ext} here to refer to the result table of query Q_j^{ext}. For convenience, we call this modified algorithm ConstructExtendedQueryGeneral.

This construction method can be extended to a more general case, where there might be multiple nested queries at the same level. In order to include the possible aggregates introduced by each nested query, we union the set of all $\{Q_i\}$ and the extended query set $\mathbf{Q}^{\mathbf{ext}}$, and call the result the *complete extended query set*, denoted by $\mathbf{Q}^{\mathbf{ext\text{-}all}}$.

Figure 2.4 shows an example. Query Q asks for all the books that are less than $25 and have more than 10 reviews. For the outmost query, Q_1^{ext} is the same as Q except for the select-list, which also includes columns referred in the where clause. Since the nested query Q_2 is correlated to the outer query, we put in Q_2^{ext} the result table of Q_1^{ext}, and replace B.isbn with Q_1^{ext}.isbn.

```
Algorithm ConstructExtendedQuery
Inputs:    query Q
Outputs:   Q^ext

begin
    step1: Copy SELECT, FROM and WHERE clauses from Q;
    step 2:Add to the select-list all attributes mentioned somewhere in Q;
           (including those appearing in any aggregate function) and the
           key for each table in the from-list. Delete from the select-list
           any aggregate functions.
    step 3:If there is no HAVING clause, go to step 5.
    step 4:Add to the WHERE clause the following constraint:
           AND (group-by-key) IN (
               SELECT      group-by-key
               (Q without the select clause)       )
    step 5:exit;
end ConstructExtendedQuery
```

Figure 2.3 Algorithm for constructing extended query

```
Q:  SELECT  isbn, title
    FROM    Books B
    WHERE   B.price < $25 AND
            10 < ( SELECT     COUNT (*)
                   FROM       Reviews R
                   WHERE      R.isbn = B.isbn )
```

Q_1^{ext}:

```
SELECT  isbn, title, price
FROM    Books B
WHERE   B.price < $25 AND
        10 < ( SELECT     COUNT (*)
               FROM       Reviews R
               WHERE      R.isbn = B.isbn )
```

Q_2^{ext}:

```
SELECT  *
FROM    Reviews R, Q1^ext
WHERE   R.isbn = Q1^ext.isbn
```

Figure 2.4 Extended query example

2.2.2.3 Queries with Views in the From Clause

Consider a query Q of the same format as in the last section, yet with another relaxation: we allow views in the FROM clause. In this case, the extended query is constructed in two phases. In the first phase, treat the views as base tables, and construct the extended query for each nested query using ConstructExtendedQueryGeneral. In phase 2, we construct the extended query set ${Q_{V_{ij}}}^{ext}$ for each view V_{ij} — the jth view at level i — using two steps. Step 1: add to V_{ij}'s WHERE clause a nested query as follows

```
AND EXISTS (  SELECT    *
              FROM      $Q_i^{ext}$
              WHERE     $Q_i^{ext}$. $V_{ij}$.key = $V_{ij}$.key        )
```

Where we implicitly assume that there is a mapping between the key of V_{ij} and the keys of the base tables of V_{ij}. The idea is to select only those rows in V_{ij} that contribute to the query result of Q_i^{ext}. Step 2: we simply regard the modified view as a regular query, and apply ConstructExtendedQueryGeneral to construct its extended query set ${Q_{V_{ij}}}^{ext}$. The union of the extended query set from phase 1 and that from phase 2 is the extended query set for query Q. The complete extended query set for Q is the union of that from each phase.

2.2.3 Specifying Currency and Consistency

We classify currency and consistency requirements into four types: *per-object*, *per-group, inter-group*, and *inter-statement*. Per-object freshness requirements, which we call *currency* *constraints***,** specify the maximal acceptable deviation for an object from its master copy. *Group consistency constraints* specify the relationship among a group of objects, for example the answers to a query. *Inter-group consistency constraints* specify the relationships among object groups, for example answer sets to multiple (sub-) queries. *Session consistency constraints* are essentially inter-group consistency constraints, but cover groups of objects arising from multiple SQL statements within a session; we do not discuss them further.

Constraints of all four types can be expressed using standard formulas constructed from object variables and constants, comparison operators, quantifiers and Boolean connectives.

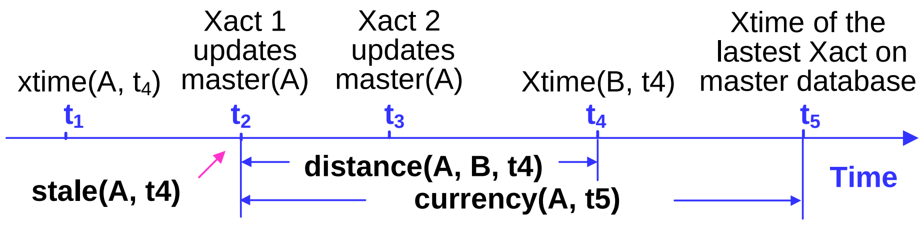


Figure 2.5 Basic concepts

2.2.3.1 Currency Constraints for Single Object

For a query Q, a user can specify currency requirements for any copy C in the complete extended query set $\mathbf{Q}^{\text{ext-all}}$ by comparing C with its counterpart in the master copy of the results of $\mathbf{Q}^{\text{ext-all}}$, in terms of either the value of C or the timestamp associated with C. In our implementation, we measure the currency of copy C in snapshot H_n by how long it has been stale, i.e., currency(C, H_n) = xtime(T_n) – stale(C, H_n).

For simplicity, we do not allow deletion on the master database. This restriction is lifted by a “ghost technique” introduced in Section 3.2.5.

Note that an attribute in the select-list might be defined by an aggregation function. For instance, if “SUM (O_TOTAL) AS TOTAL” appears in the select-list, a user can specify requirements on this aggregate attribute TOTAL. However, such derived data does not have a master copy; hence its currency is not well-defined. We remedy this by a “virtual master copy” technique introduced in Section 3.2.6.

2.2.3.2 Group Consistency for Cached Objects

The function **return**(O, s) returns the value of O in database state s. We say that object O in s_{cache} is **snapshot consistent** with respect to a database snapshot H_n if **return**(O, s_{cache}) = **return**(O, H_n) and xtime(O, H_n) = xtime(master (O), H_n).

Given how copies are updated through copy transactions, we observe that for every object in a cache, there is at least one database snapshot (the one with which it was synchronized) with respect to which it is snapshot consistent. However, different objects in a cache could be consistent with respect to different snapshots. For a subset K of the cache, if a snapshot H_n exists such that each object in K is snapshot consistent with respect to H_n, then we say K is **snapshot consistent** with respect to H_n. If K is the entire cache, we say that the cache is **snapshot consistent**.

We define the distance between two objects (which could be masters or copies) A and B in a snapshot H_n as follows. Let xtime(B, H_n) = T_m and let xtime(A, H_n) $\leq$ xtime(B, H_n). Then:

distance(A, B, H_n) = currency(A, H_m)

Since B is current (identical to its master) at time $T_{m,}$ the distance between A and B reflects how close A and B are to being snapshot consistent with respect to snapshot H_m. Figure 2.5 illustrates the basic concepts.

Let t be the distance between A and B. We say that A and B are **Δ-consistent** with consistency bound t. We also extend the notion of Δ-consistency for a set of objects K, by defining the bound t to be the maximum distance between any pair of objects in K.

Consider a set of cached objects K and a database snapshot H_n. If K is Δ-consistent with consistency bound t = 0, and O is the object with the largest value of xtime(O, H_n) in K, it is easy to show that K is snapshot-consistent with respect to the database snapshot at xtime(O, H_n). In general, as t increases, the deviation from snapshot consistency also increases.

2.2.3.3 Group Consistency for Queries

Given a query Q, the relevant set for Q (the result of the extended version Q^{ext}) includes all objects that affect the result of Q. We can apply the concept of Δ-consistency to this set, and thereby impose a consistency constraint on Q.

In practice, however, we may not care whether the entire relevant set is Δ-consistent, and simply wish to require that certain subsets of the relevant set be Δ-consistent. We leverage the power of SQL queries to achieve this, as follows. Given query Q, we allow the use of an auxiliary set of queries P over the relevant set of Q to identify the subset that must be Δ-consistent. We illustrate the approach by discussing two common cases.

1) **Consistency requirements on input tables of query Q**: We may want to state that one or more input tables must be from a single database snapshot. We can do this using a query p that simply selects all attributes associated with those tables from Q^{ext} and requiring Δ-consistency with respect to the result of p.

2) **Consistency with respect to horizontal partitions of the result of query Q:** Again, we use an auxiliary query p over Q^{ext}. We can use SQL's GROUP BY clause to divide the result of p horizontally into partitions, and require Δ-consistency with respect to one or more partitions (selected using the HAVING clause).

2.2.3.4 Inter-Group Consistency

We have discussed two natural ways in which groups of related objects arise, namely as subsets of a cache, or as part of the result of a query. It is sometimes necessary to impose consistency requirements across multiple groups of objects. Examples include:

1) Multiple groups of cached objects, such as all cached Order records and all cached Catalog records.

2) Groups of objects from different blocks of a query. (Note that each subquery has an extended version!)

3) Groups of objects drawn from multiple statements (e.g., different queries) within a session.

Regardless of the context in which groups arise, let G1, G2, … , Gn be the sets of relevant data objects for groups 1 to n.

A user can specify two types of consistency requirements over this collection:

Δ-consistency: Naturally, we can require that the objects in the union or intersection of one or more groups be Δ-consistent with bound t.

Time-line consistency: Intuitively, we might want to say that "time always moves forward" across a certain ordering of groups. That is, for any i, j such that $i < j \leq n$, any objects A $\in$ Gi, B $\in$ Gj, xtime(A, H_n) ≤ xtime(B, H_n), where H_n is the database snapshot after executing all statements corresponding to the groups G1, G2, … , Gn.

2.2.3.5 Session Consistency Constraints

We extend the previous concept to a group of statements. Given a group of ordered queries Q_1, …, Q_n, similar to the single query case, we allow a user to use a set of auxiliary queries P_i over Q_i^{ext}, the relevant set of Q_i, and specify *Δ-consistency* or *time-line consistency* requirements over any subset of $\cup$ (P_1..P_n). While Δ-consistency constraints bound the divergence of the result unit set from one snapshot, time-line constraints require time to move forward within the group with regards to the specified result unit sets.

Chapter 3

Data Quality-Centric Caching Model

SQL extensions that allow individual queries to explicitly specify data quality requirements in terms of currency and consistency were proposed in Chapter 2. This chapter develops a data quality-aware, fine-grained cache model and studies cache design in terms of four fundamental properties: *presence, consistency, completeness* and *currency*. Such a model provides an abstract view of the cache to the query processing layer, enabling the implementation of the latter to be independent of the former. Regardless of the cache management mechanisms or policies used, as long as cache properties are observed, query processing can deliver correct results. Further, the flexible mechanism provided by this cache model opens the door for adaptive cache management.

3.1 Introduction

3.1.1 Background

We model cached data as materialized views over a master database. Queries can specify their data quality requirements in terms of currency and consistency. It is the caching DBMS's responsibility to ensure that it produces query results that satisfy the stated quality requirements. In order to do that, the caching DBMS needs to keep track of local data quality. The question is, at what granularity? At one end of the spectrum is database level consistency. That is, the whole cache has to be consistent. This approach requires the least bookkeeping, but is the least flexible: suppose a query requires an object to be less than 1 second stale, if we want to answer that query locally, we have to keep the whole cache no more than 1 second stale.

A slightly refined granularity is view level. That is, all rows of a cached view are consistent, i.e., from the same database snapshot. We explain the corresponding maintenance and query processing mechanisms for this granularity level in Chapter 4.

Although more flexible than the database level granularity, view level granularity still severely restricts the cache maintenance policies that can be used. A pull policy, where the cache explicitly refreshes data by issuing queries to the source database, offers the option of using query results as the units for maintaining consistency and other cache properties. In particular, issuing the same parameterized query with different parameter values returns different partitions of a cached view, offering a much more flexible unit of cache maintenance (view partitions) than using entire views. This is the focus of this chapter. Figure 3.1 shows our running example, where Q1 is a parameterized query, followed by different parameter settings.

Authors (authorId, name, gender, city, state)

Books (isbn, authorId, publisherId, title, type)

```
Q1:  SELECT    *
     FROM      Authors A
     WHERE     authorId in (1,2,3)
     CURRENCY BOUND 10 min on (A) BY $key

E1.1: $key = Ø
E1.2: $key = authorId
E1.3: $key = city

Q2:  SELECT    *
     FROM      Authors A, Books B
     WHERE     authorId in (1,2,3) AND A.authorId = B.authorId
     CURRENCY BOUND 10 min ON (A, B) BY authorId

Q3:  SELECT * FROM Authors A WHERE city = "Madison"
     CURRENCY BOUND 10 min ON (A) BY authorId
```

Figure 3.1 Running examples

3.1.2 Motivation

We now motivate four properties of cached data that determine whether it can be used to answer a query. In the model proposed in Chapter 2, a query's C&C constraints are stated in a currency clause. For example, in Q2, the currency clause specifies three "quality" constraints on the query results: 1) "ON (A, B)" means that all Authors and Books rows returned must be *consistent*, i.e., from the same database snapshot. 2) "BOUND 10 min" means that these rows must be *current* to within 10 minutes, that is, at most 10 minutes out of date. 3) “BY authorId” means that all result rows with the same authorId value must be consistent. To answer the query from cached data, the cache must guarantee that the result satisfies these requirements and two more: 4) the Authors and Books rows for authors 1, 2, and 3 must be *present* in the cache and 5) they must be *complete*, that is, no rows are missing.

E1.1 requires that all three authors with id 1, 2 and 3 be present in the cache, and that they be mutually consistent. Suppose we have in the cache a partial copy of the Authors table, AuthorCopy, which contains some frequently accessed authors, say those with authorId 1-10. We could require the cache to guarantee that all authors in AuthorCopy be mutually consistent, in order to ensure that we can use the rows for authors with id 1, 2 and 3 to answer E1.1, if they are present. However, query E1.1 can be answered using the cache as long as authors 1, 2 and 3 are mutually consistent, regardless of whether other author rows are consistent with these rows. On the other hand, if the cache provides no consistency guarantees, i.e., different authors could have been copied from a different snapshot of the master database, the query cannot be answered using the cache even if all requested authors are present. In contrast, query E1.2, in which the BY clause only requires rows for a given author to be consistent, can be answered from the cache in this case.

Query Q3 illustrates the completeness property. It asks for all authors from Madison, but the rows for different authors do not have to be mutually consistent. Suppose we keep track of which authors are in the cache by their authorIds. Even if the cache happens to contain all the authors from Madison, we cannot safely use the cached data unless the cache guarantees

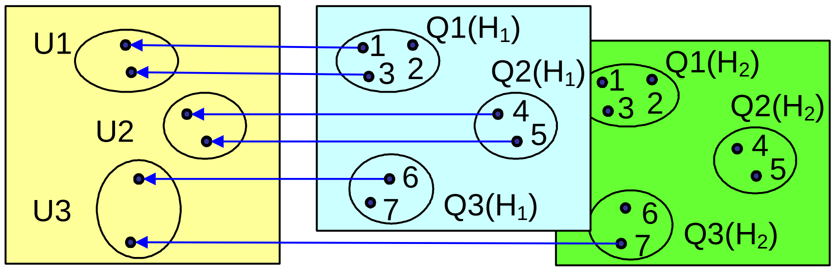


Figure 3.2 Cache property example

that it has all the authors from Madison. Intuitively, the cache guarantees that its content is complete w.r.t. the set of objects in the master database that satisfy a given predicate.

Regardless of the cache management mechanisms or policies used, as long as cache properties are observed, query processing can deliver correct results. Thus, cache property descriptions serve as an abstraction layer between query processing and cache management, enabling the implementation of the former to be independent of the latter.

The rest of the chapter is organized as follows. Section 3.2 builds a solid foundation for cache description by formally defining presence, consistency, completeness and currency. Section 3.3 introduces a novel cache model that supports a specific way of partitioning the cache and translating a rich class of integrity constraints (expressed in extended SQL DDL syntax) into properties required to hold over different partitions. We identify an important property of cached views, called *safety*, and show how safety aids in efficient cache maintenance in Section 3.4. Finally, we formally define cache schemas and characterize when they are safe, offering guidelines for cache schema design in Section 3.5.

3.2 Formal Definition of Cache properties

3.2.1 Presence

The simplest type of query asks for an object identified by its key (e.g., Q1). How to tell if an object is in the cache?

Intuitively, we require every object in the cache to be copied from some valid snapshot. Let **return**(O, s) return the value of object O in database state s. We say that copy C in a cache state S_{cache} is **snapshot consistent** w.r.t. a snapshot H_n of the master database if return(C, S_{cache}) = return(master(C), H_n) and xtime(C, H_n) = xtime(master(C), H_n). We also say **CopiedFrom**(C, H_n) holds.

Definition: (**Presence**) An object O is present in cache S_{cache} iff there is a copy C in S_{cache} s.t. master(C) = O, and for some master database snapshot H_n CopiedFrom(C, H_n) holds. □

3.2.2 Consistency

When a query asks for more than one object, it can specify mutual consistency requirements on them, as shown in E1.1.

For a subset U of the cache, we say that U is mutually snapshot consistent (consistent for short) w.r.t. a snapshot H_n of the master database iff CopiedFrom(O, H_n) holds for every object O in U. We also say CopiedFrom(U, H_n) holds.

Besides specifying a consistency group by object keys (e.g., authorId in E1.2), a query can also specify a consistency group by a selection, as in E1.3. Suppose all authors with id 1, 2 and 3 are from Madison. The master database might contain other authors from Madison. The cache still can be used to answer this query as long as all three authors are mutually consistent and no more than 10 minutes stale. Given a query Q and a database state s, let Q(s) denote the result of evaluating Q on s.

Definition: (**Consistency**) For a subset U of the cache S_{cache}, if there is a snapshot H_n of the master database s.t. CopiedFrom(U, H_n) holds, and for some query Q, $U \subseteq Q(H_n)$, then U is snapshot consistent (or consistent) w.r.t. Q and H_n. □

U consists of copies from snapshot H_n and Q is a selection query. Thus the containment of U in $Q(H_n)$ is well defined. Note that object metadata, e.g., timestamps, are not used in this comparison.

If a collection of objects is consistent, then any of its subsets is also consistent. Formally,

Lemma 3.1: If a subset U of the cache S_{cache} is consistent w.r.t. a query Q and a snapshot H_n, then subset P(U) defined by any selection query P is consistent w.r.t. P°Q and H_n. □

Proof of Lemma 3.1:

Since U is consistent w.r.t. Q and H_n, we have:

$$U \subseteq Q(H_n) \qquad (1)$$

$$\text{CopiedFrom}(U, H_n) \qquad (2)$$

Since (1), for any selection query P,

$$P(U) \subseteq P°Q\,(H_n) \qquad (3)$$

Since P is a selection query, $P(U) \subseteq U$. Together with (2), we have

CopiedFrom(P(U), H_n) (4)

From (3) and (4), we know that P(U) is snapshot consistent w.r.t. P°Q and H_n. □

3.2.3 Completeness

As illustrated in Q3, a query might ask for a set of objects defined by a predicate. How do we know that *all* the required objects are in the cache?

Definition: (**Completeness**) A subset U of the cache S_{cache} is complete w.r.t. a query Q and a snapshot H_n of the master database iff CopiedFrom**(**U, H_n) holds and U = Q(H_n). □

Lemma 3.2: If a subset U of the cache S_{cache} is complete w.r.t. a query Q and a snapshot H_n, then subset P(U) defined by any selection query P is complete w.r.t. P°Q and H_n. □

Proof of Lemma 3.2:

From the given, we have

CopiedFrom(U, H_n) (1)

U = Q(H_n) (2)

From (2), for any selection query P,

P(U) = P°Q(H_n) (3)

Since P(U) $\subseteq$ U, from (1), we have

CopiedFrom(P(U), H_n) (4)

From (3) and (4), we know P(U) is complete w.r.t. P°Q and H_n. □

The above constraint is rather restrictive. Assuming that objects' keys are not modified, it is possible to allow subsequent updates of some objects in U to be reflected in the cache, while still allowing certain queries (which require completeness, but do not care about the modifications and can therefore ignore consistency) to use cached objects in U.

Definition: (**Associated Objects**) We say that a subset U of the cache S_{cache} is associated with a query Q if for each object C in U, there exists a snapshot H_n of the master database such that CopiedFrom(C, H_n) holds and C is in Q(H_n). □

Definition: (**Key-completeness**) For a subset U of the cache S_{cache}, we say U is key-complete w.r.t. Q and a snapshot H_n, iff U is associated with Q and $\Pi_{key}Q(H_n) \subseteq \Pi_{key}(U)$. □

Intuitively, U includes (as identified by the keys) all the objects that appear in the result of Q applied to the master database H_n. However, the objects in the cache might have been copied from different earlier snapshots of the master database, and subsequent changes to these objects might not be reflected in the cache.

Figure 3.2 illustrates cache properties, where an edge from object O to C denotes that C is copied from O. Assuming all objects are modified in H_2, U1 is consistent but not complete w.r.t. Q1 and H_1, U2 is complete w.r.t. Q2 and H_1, and U3 is key-complete w.r.t. Q3 and both H_1 and H_2.

Lemma 3.3: If a subset U of the cache S_{cache} is complete w.r.t. a query Q and a database snapshot H_n, then U is both key-complete and consistent w.r.t. Q and H_n. □

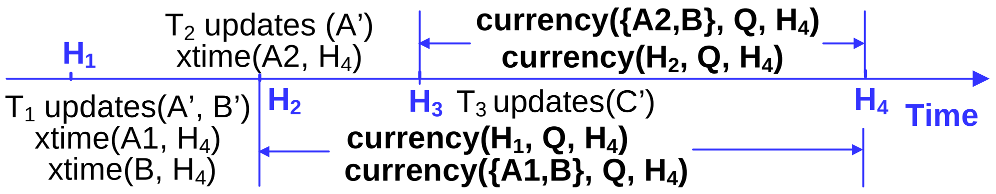


Figure 3.3 Currency example (1)

Proof of Lemma 3.3:

Directly from the definitions. □

3.2.4 Currency

We have defined *stale point* and *currency* for a single object. Now we extend the concepts to a set of objects. Suppose that at 1pm, there are only two authors from Madison in the master database, and we copy them to the cache, forming set U. At 2pm, a new author moves to Madison. At 3pm, how stale is U w.r.t. predicate "city = Madison"? Intuitively, the answer should be 1 hour, since U gets stale the moment the new author is added to the master database. However, we cannot use object currency to determine this since both objects in U are still current. For this reason we use the snapshot where U is copied from as a reference.

We overload stale() to apply to a database snapshot H_m w.r.t. a query Q: **stale**(H_m, Q, H_n) is equal to xtime(A), where A is the first transaction that changes the result of Q after H_m in H_n. Similarly, we overload the currency() function: **currency**(H_m, Q, H_n) = xtime(H_n) – stale(H_m, Q, H_n).

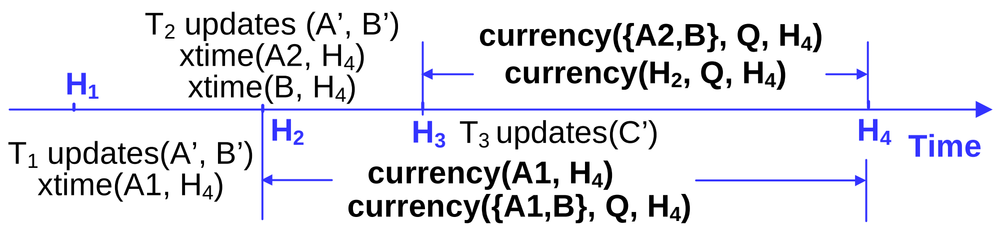


Figure 3.4 Currency example (2)

Definition: (**Currency for complete set**) If a subset U of the cache S_{cache} is complete w.r.t. a query Q and a snapshot H_m, then the currency of U w.r.t. a snapshot H_n of the master database is: currency(U, Q, H_n) = currency(H_m, Q, H_n). □

From the definition, the currency of U depends on the snapshot H_m used in the calculation. This problem can be solved using a "ghost row" technique, see Section 3.2.5 for details.

Figure 3.3 illustrates the currency of two complete sets, where A1 and A2 are two copies of A' and B is a copy of B', $Q(H_i)$ = {A', B'}, i = 1, 2, $Q(H_i)$ = {A', B', C'}, i = 3, 4. {A1, B} and {A2, B} are complete w.r.t. Q and H_1, H_2.

Non-Shrinking Assumption: For any query Q, any database snapshot H_i and H_j, where $i \leq j$, $\Pi_{key}Q(H_i) \subseteq \Pi_{key}Q(H_j)$. □

Currency Property 3.1: Under the assumption above, for any subset U of the cache S_{cache}, any query Q, and any master database snapshots H_i and H_j, if U is complete w.r.t. Q and both H_i and H_j, then for any n, currency(H_i, Q, H_n) = currency(H_j, Q, H_n). □

Proof of Currency Property 3.1:

(By contradiction)

Since the case i=j is trivial, without loss of generality, assume i<j. Assume T_k is the first transaction that modifies Q(Hi) after H_i. We claim that k>j. For the proof by contradiction, assume k≤j.

From the non-shrinking assumption, Tk either 1) modifies an object in Q(H_i), say O1 or 2) adds a new object, say O2 to the result of Q. Further, both O1 and O2 are in Q(Hj).

In case 1), since k≤j, xtime(O1, H_j)>xtime(O1, H_i), which contradicts the given that U is consistent w.r.t. both H_i and H_j.

In case 2), O2 is not in Q(H_i), which also contradicts the given that U is complete w.r.t. both H_i and H_j.

Thus k>j, hence currency(H_i, Q, H_n) = currency(H_j, Q, H_n). □

How to measure the currency of a key-complete set? Figure 3.4 shares the same assumptions as Figure 3.3, except for T_2 and xtime(B), where {A1, B}and {A2, B} are key-complete w.r.t. Q and H_1 and H_2, while the latter is also complete w.r.t. Q and H_2. It is desirable that 1) currency({A1,B}, Q, H_4) is deterministic; and 2) since A1 is older than A2, {A1, B}should be older than {A2, B}.

We address these problems by firstly identifying a unique referenced snapshot, and secondly incorporating the currency of the objects into the currency definition.

Definition: (**Max key-complete snapshot**) For any subset U of the cache S_{cache} and a query Q, the max key-complete snapshot of U w.r.t. Q and a database snapshot H_n, **max-snapshot**(U, Q, H_n) is equal to H_k, if there exists k, s.t., for any i≤k, $\prod_{key} Q(H_i) \subseteq \prod_{key} U$, and one of the following conditions holds: 1) k=n; 2) $\prod_{key} U \subset \prod_{key} Q(H_{k+1})$. Otherwise it is Ø. □

Directly from the definition of key-completeness and the non-shrinking assumption, we have the following lemma.

Lemma 3.4: If there exists a database snapshot H_m, s.t. U is key-complete w.r.t. Q and H_m, then for any n, max-snapshot(U, Q, H_n) is not Ø. □

Lemma 3.4 guarantees that the following definition is well defined for a key-complete set.

Definition: (**Currency for key-complete set**) For a subset U of the cache S_{cache}, if U is key-complete w.r.t. a query Q and some database snapshot, then the currency of U w.r.t. a snapshot H_n of the master database is defined as follows. Let H_m = max-snapshot(U, Q, H_n) and

$$Y = \max_{C \in U} (currency\ (C, H_n)),$$

Then currency(U, Q, H_n) = max (Y, currency(H_m, Q, H_n)). □

Figure 3.4 shows the currency of a key-complete set {A1, B} and a complete set {A2, B}.

The currency of a key-complete set has some nice properties that intuitively fit.

Currency Property 3.2: For any subset U of the cache S_{cache}, and a query Q, if U is key-complete w.r.t. Q and some database snapshot, then for any n, currency(U, Q, H_n) is deterministic. □

Proof of Currency Property 3.2:

Directly from the definition and Lemma 3.4. □

Currency Property 3.3: Given any query Q, and two subsets U1 and U2 of the cache S_{cache}, if max-snapshot(U1, Q, H_n) = max-snapshot(U2, Q, H_n) ≠ Ø, let

$$Y_i = \max_{O \in U_i} (currency(O, H_n)),$$

where i=1, 2. If $Y_1 \geq Y_2$, then currency(U1, Q, H_n) ≥ currency(U2, Q, H_n). □

Proof of Currency Property 3.3:

Directly from the definition. □

Currency Property 3.4: currency-complete is a special case of currency-key-complete. □

Proof of Currency Property 3.4:

Given any subset U of the cache S_{cache} that is complete w.r.t. a query Q and some database snapshot H_m. For any n≥m, let H_g = max-snapshot(U, Q, H_n). From the definition of max key-complete snapshot we know g≥m. There are two cases:

Case 1: U is complete w.r.t. H_g.

Let T_k be the first transaction in H_n that changes the result of Q after H_g. From the non-shrinking assumption, again, we have two cases:

Case 1.1: T_k touches at least one object, say O1, in U. Since T_k is the first transaction that touches U,

$$Y = \max_{O \in U} (currency\ (O, H_n)) = currency\ (O_1, H_n) \qquad (1)$$

Since the stale points for O1 and $Q(H_g)$ are both $xtime(T_k)$, $currency(H_g, Q, H_n) = currency(O1, H_n)$. Thus

$$currency(U, Q, H_n) = \max (Y, currency(H_g, Q, H_n))$$
$$= currency(H_g, Q, H_n) = currency(O1, H_n).$$

Case 1.2: T_k adds new objects into the result of Q.

In this case the stale point of any object O in U is later than $xtime(T_k)$, so $currency(H_g, Q, H_n) \geq currency(O, H_n)$.

$$currency(U, Q, H_n) = \max (Y, currency(H_g, Q, H_n))$$
$$= currency(H_g, Q, H_n).$$

Case 2: U is not complete w.r.t. H_g.

Let T_k be the first transaction in H_n that modifies at least an object, say O1 in U after H_m, then

$$currency(H_m, Q, H_n) = currency(O1, H_n) \qquad (2)$$

$$Y = \max_{O \in U} (currency\ (O, H_n)) = currency\ (O_1, H_n) \qquad (3)$$

In addition we have $k \leq g$, otherwise from the non-shrinking assumption, U would be complete w.r.t. H_g. Thus

$$Y \geq \text{currency}(H_g, Q, H_n) \qquad (4)$$

Putting (2), (3) and (4) together,

$$\text{currency}(U, Q, H_n) = \max (Y, \text{currency}(H_g, Q, H_n))$$
$$= \text{currency}(H_g, Q, H_n) = \text{currency}(O1, H_n). \quad \square$$

3.2.5 Dealing with Deletion

Currency properties 2.1 to 2.4 don't hold without the non-shrinking assumption. Take Property 2.1 for example. On day 1 there are two customers C1, C2 from WI, which we copied to the cache, U = {C1, C2}. On day 2, customer C3 moved to WI temporarily, and moved out of WI on day 5. Then on day 4, the currency of U is 2 days stale. However, on day 6, it goes back to 0!

The reason is that when an object is deleted, we lose its xtime record. Consequently, given a set of objects **K**, one cannot uniquely identify the first snapshot **K** appears in. To remedy that, we introduce the concept of *ghost object*. Conceptually, when an object is deleted from a region in the master copy, we don't really delete it, instead, we mark it as a ghost object and treat it the same way as a normal object. Thus we keep the xtime timestamp of deleted objects. Ghost objects and their timestamps are propagated to the cache just as normal objects. With this technique, deletion is modeled as a special modification. Thus the non-shrinking assumption is guaranteed even in the presence of deletions.

Lemma 3.5: With the ghost object technique, given any query Q, the non-shrinking assumption holds. □

Proof of Lemma 3.5:

With the ghost object technique, there are no deletions to the region defined by Q. □

Note that in practice, we don't need to record those ghost objects, since the calculation of currency only needs to be conservative. How we bound the currency of a complete set is discussed in Section 3.4.1.2.

3.2.6 Derived Data

If the cache only contains (parts of) base tables, then for each object in the cache there is a master version in the master database. This doesn't apply to derived data, i.e., materialized views in the cache. An object (row) in a materialized view in the cache doesn't necessarily have a master copy in the master database. We introduce the concept of *virtual master copy* to remedy this. Conceptually, for any view V in the cache, for any snapshot H_i of the master database, we calculate $V(H_i)$ and include it in the master database. Thus, by comparing two adjacent snapshots, we can record any insertion/deletion/modification on the view. With this technique, any object in the cache — no matter whether it is from a base table or a view — has a master copy in the master database. Thus, any query can be used to define a region in the cache.

Again, in practice, since we only need to bound the currency of a region conservatively, we don't need to materialize the virtual master copies. See Section 3.4.1.2.

3.3 Dynamic Caching Model

In our model, a cache is a collection of materialized views **V** = {V_1, …, V_m}, where each

```
D1:  CREATE VIEW AuthorCopy AS
            SELECT * FROM Authors

     CREATE VIEW BookCopy AS
            SELECT * FROM Books

D2:  CREATE TABLE AuthorList_PCT(authorId int)
     ALTER VIEW AuthorCopy
            ADD PRESENCE ON authorId IN
                (SELECT authorId FROM AuthorList_PCT)

D3:  CREATE TABLE CityList_CsCT(city string)
     ALTER VIEW AuthorCopy
            ADD CONSISTENCY ON city IN
                (SELECT city FROM CityList_CsCT)

D4:  CREATE TABLE CityList_CpCT(city string)
     ALTER VIEW AuthorCopy
            ADD COMPLETE ON city IN
                (SELECT city FROM CityList_CpCT)

D5:  ALTER VIEW BookCopy
     ADD PRESENCE ON authorId IN
                (SELECT authorId FROM AuthorCopy)

D6:  ALTER VIEW BookCopy ADD CONSISTENCY ROOT
```

Figure 3.5 DDL examples for adding cache constraints

view V_i is defined using a query expression Q_i. We describe the properties of the cache in terms of integrity constraints defined over **V**. In this section, we introduce a class of metadata tables called *control tables* that facilitate specification of cache integrity constraints, and introduce extended SQL DDL syntax for constraint specification. Figure 3.5 shows the set of DDL examples used in this section. We start by defining two views as shown in D1.

3.3.1 View Partitions and Control-tables

Instead of treating all rows of a view uniformly, we allow them to be partitioned into smaller groups, where properties (presence, currency, consistency or completeness) are guaranteed per group. The same view may be partitioned into different sets of groups for different properties. Further, the cache may provide a *full* or *partial guarantee*, that is, it may guarantee that the property holds for all groups in the partitioning or only for some of the groups. Although different implementation mechanisms might be used for full and partial guarantees, conceptually, the former is a special case of the latter; we therefore focus on partial guarantees.

In this thesis, we impose restrictions on how groups can be defined and consider only groups defined by equality predicates on one or more columns of the view. That is, two rows belong to the same group if they agree on the values of the grouping columns. For a partial guarantee, the grouping values for which the guarantee holds are (conceptually) listed in a separate table called a **control table**. Each value in the control table corresponds to a group of rows of V_i that we call a **cache region** (or simply **region**). Each view V_i in **V** can be associated with three types of control tables: *presence*, *consistency* and *completeness control tables*. We use *presence/consistency/completeness region* to refer to cache regions defined for each type respectively. Note that control tables are conceptual; some might be explicitly maintained and others might be implicitly defined in terms of other cached tables in a given implementation.

3.3.1.1 Presence Control-Table (PCT)

Suppose we receive many queries looking for some authors, as in Q1. Some authors are much more popular than others and the popular authors change over time, i.e., the access pattern is skewed and changes over time. We would like to answer a large fraction of queries locally but maintenance overhead are too high to cache the complete Authors table. Further, we want to be able to adjust cache contents for the changing workload without changing the view definition. These goals are achieved by presence control tables.

A **presence control table (PCT)** for view V_i is a table with a 1-1 mapping between a subset K of its columns and a subset K' of V_i's columns. We denote this by PCT[K, K']; $K \subseteq PCT$ is called the **presence control-key** (**PCK**) for V_i, and $K' \subseteq V_i$ is called the **presence controlled-key (PCdK)**. For simplicity, we will use PCK and PCdK interchangeably under the mapping. A PCK defines the smallest group of rows (i.e., an object) that can be admitted to or evicted from the cache in the MTCache "pull" framework. We assume that the cache maintenance algorithms materialize, update and evict all rows within such a group together.

Presence Assumption: All rows associated with the same presence control-key are assumed to be present, consistent and complete. That is, for each row s in the presence control table, subset U = $\sigma_{K'=s.K}$ (V_i) is complete and thus consistent w.r.t. ($\sigma_{K'=s.K} \circ Q_i$) and H_n, for some snapshot H_n of the master database, where Q_i is the query that defines V_i . □

If V_i has at least one presence control table, it is a **partially materialized view (PMV)**, otherwise it is a fully materialized view. See [ZLG05] for more general types of partial views, partial view matching, and run-time presence checking.

In our motivating example, we cache only the most popular authors. This scenario can be handled by creating a presence control table and adding a PRESENCE constraint to AuthorCopy, as in D2. AuthorList_PCT acts as a presence control table and contains the ids of the authors who are currently present in the view AuthorCopy, i.e., materialized in the view.

3.3.1.2 Consistency Control-Table (CsCT)

A local view may still be useful even when all its rows are not kept mutually consistent, e.g., in a scenario where we receive many queries like E1.3. Suppose AuthorCopy contains all the required rows. If we compute the query from the view, will the result satisfy the query's consistency requirements? The answer is "not necessarily" because the query requires all result rows to be mutually consistent per city, but AuthorCopy only guarantees that the rows for each author are consistent; nothing is guaranteed about authors from a given city. The consistency control table provides the means to specify a desired level of consistency.

A **consistency control table (CsCT)** for view V_i is denoted by CsCT[K], where a set of columns $K \subseteq$ CsCT is also a subset of V_i, and is called the **consistency control-key** (**CsCK**) for V_i. For each row s in CsCT, if there is a row t in V_i, s.t. s.K = t.K, then subset $U = \sigma_{K=s.K}(V_i)$ must be consistent w.r.t. $(\sigma_{K=s.K} \circ Q_i)$ and H_n for some snapshot H_n of the master database.

In our example, it is desirable to guarantee consistency for all authors from the same city, at least for some of the popular cities. We propose an additional CONSISTENCY constraint for specifying this requirement. We first create a consistency control table containing a set of cities and then add a CONSISTENCY constraint to AuthorCopy, as in D3 of Figure 3.5. The CONSISTENCY clause specifies that the cache must keep all rows related to the same city consistent if the city is among the ones listed in CityList_CsCT; this is in addition to the consistency requirements implicit in the Presence Assumption. AuthorCopy can now be used to answer queries like E1.3.

If we want the cache to guarantee consistency for every city, we change the clause to CONSISTENCY ON city. If we want the entire view to be consistent, we change the clause to CONSISTENCY ON ALL. If we don’t specify a consistency clause, the cache will not provide any consistency guarantees beyond the minimal consistency implied by the presence control table under the Presence Assumption.

3.3.1.3 Completeness Control-Table (CpCT)

A view with a presence control table can only be used to answer point queries with an equality predicate on its control columns. For example, AuthorCopy cannot answer Q3.

It is easy to find the rows in AuthorCopy that satisfy the query but we cannot tell whether the view contains ***all*** required rows. If we want to answer a query with predicate P on columns other than the control-keys, the cache must guarantee that all rows defined by P appear in the cache or none appear. Completeness constraints can be expressed with completeness control tables.

A **completeness control table (CpCT)** for view V_i is denoted by CpCT[K]. A completeness control table is a consistency control table with an additional constraint: the subset U in V_i defined as before is not only consistent but also complete w.r.t. $(\sigma_{K=s.K} \circ Q_i)$ and H_n, for some snapshot H_n of the master database. We say K is a **completeness control-key** (**CpCK**). Note that all rows within the same completeness region must also be consistent (Lemma 3.3).

We propose to instruct the cache about completeness requirements using a COMPLETENESS constraint. Continuing our example, we create a completeness control table and then add a completeness clause to the AuthorCopy definition, as in D4 of Figure 3.5. Table CityList_CpCT serves as the completeness control table for AuthorCopy. If a city is contained in CityList_CpCT, then we know that either all authors from that city are contained in AuthorCopy or none of them are. Note that an entry in the completeness control table does not imply presence. Full completeness is indicated by dropping the clause starting with “IN”. Not specifying a completeness clause indicates that the default completeness implicit in the Presence Assumption is sufficient.

A similar property is termed “domain completeness” in DBCache [ABK+03]. However, our mechanism provides more flexibility. The cache admin can specify: 1) the subset of columns to be complete; 2) to force completeness on all values or just a subset of values for these columns.

3.3.2 Correlated Presence Constraints

In our running example, we may not only receive queries looking for some authors, but also follow-up queries looking for related books. That is, the access pattern to BookCopy is decided by the access pattern to AuthorCopy. In order to capture this, we allow a view to use another view as a presence control table. To have BookCopy be controlled by AuthorCopy, we only need to declare AuthorCopy as a presence control table by a PRESENCE constraint in the definition of BookCopy, as in D5 of Figure 3.5.

If a presence control table is not controlled by another one, we call it a **root presence control table**. Let **L** = {V_{m+1}, …, V_n} be the set of root presence control tables; **W** = **V** $\cup$ **L**. We depict the presence correlation constraints by a **cache graph**, denoted by **<W**, **E>**. An edge $V_i \xrightarrow{K_{i,j}, K_{i,j}'} V_j$ means that V_i is a PCT[$K_{i,j}$, $K_{i,j}$ ’] of V_j.

Circular dependencies require special care in order to avoid “unexpected loading”, a problem addressed in [ABK+03]. In our model, we don’t allow circular dependencies, as stated in Rule 1 in Figure 3.11. Thus we call a cache graph a **cache DAG**.

Each view in the DAG has two sets of orthogonal properties. First, whether it is view-level or group-level consistent. Second, to be explained shortly, whether it is consistency-wise correlated to its parent. For illustration purposes, we use shapes to represent the former: circles for view-level consistent views and rectangles (default) for all others. We use colors to denote the latter: gray if a view is consistency-wise correlated to its parents, white (default) otherwise.

Definition: (**Cache schema)** A cache schema is a cache DAG **<W**, **E>** together with the completeness and consistency control tables associated with each view in **W**. □

3.3.3 Correlated Consistency Constraints

In our running example, we have an edge AuthorCopy $\xrightarrow{authorId}$ BookCopy, meaning if we add a new author to AuthorCopy, we always bring in all of the author's books. The books

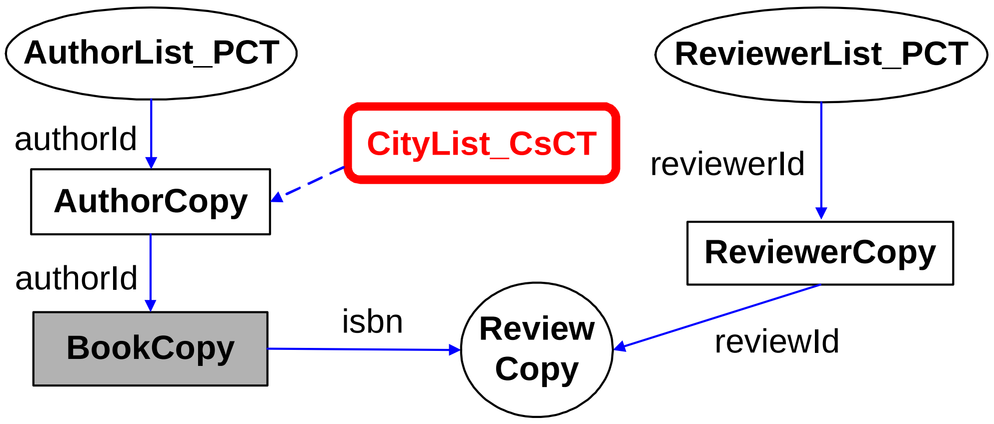


Figure 3.6 Cache schema example

of an author have to be mutually consistent, but they are not required to be consistent with the author.

If we wish the dependent view to be consistent with the controlling view, we add the consistency clause: CONSISTENCY ROOT, as in D6 of Figure 3.5. A node with such constraint is colored *gray*; it cannot have its own consistency or completeness control tables (Rule 2 in Figure 3.11).

For a gray node V, we call its closest white ancestor its **consistency root**. For any of V's cache regions U_j, if U_j is controlled by a PCK value included in a cache region U_i in its parent, we say that U_i **consistency-wise controls** U_j; and that U_i and U_j are **consistency-wise correlated**.

Figure 3.6 illustrates a cache schema example, which consists of four partially materialized views. AuthorCopy is controlled by a presence control table AuthorList_PCT, likewise for ReviewerCopy and ReviewerList_PCT. Besides a presence control table, AuthorCopy has a consistency control table CityList_CsCT on city. BookCopy is both presence-wise and consistency-wise correlated to AuthorCopy. In contrast, ReviewCopy has two presence control tables: BookCopy and ReviewerCopy; it is view level consistent and consistency-wise independent from its parents.

3.4 Safe Cached Views

A cache has to perform two tasks: 1) populate the cache and 2) reflect updates to the contents of the cache, while maintaining the specified cache constraints. Complex cache constraints can lead to unexpected additional fetches in a pull-based maintenance strategy, causing severe performance problems. We illustrate the problems through a series of examples, and quantify the refresh cost for unrestricted cache schemas in Theorem 3.1. We then identify an important property of a cached view, called *safety* that allows us to optimize pull-based maintenance, and summarize the gains it achieves in Theorem 3.2. We introduce the concept of *ad-hoc* cache regions, used for adaptively refreshing the cache.

For convenience, we distinguish between the schema and the instance of a cache region U. The schema of U is denoted by <V, K, k>, meaning that U is defined on view V by a control-key K with value k. We use the *italic* form *U* to denote the instance of U.

3.4.1 Pull-Based Cache Maintenance

In the "pull" model, we obtain a consistent set of rows using either a single query to the backend or multiple queries wrapped in a transaction. As an example, suppose AuthorCopy, introduced in Section 3.3, does not have any children in the cache DAG and that the cache needs to refresh a row t (1, Rose, Female, Madison, WI).

First, consider the case where AuthorCopy does not have any consistency or completeness control table, and so consistency follows the presence table. Then all rows in the presence region identified by authorId 1 need to be refreshed together. This can be done by issuing the presence query shown in Figure 3.8 to the backend server.

```
Presence query:          Consistency query:        Completeness query:

SELECT   *               SELECT  *                 SELECT  *
FROM     Authors         FROM    Authors           FROM    Authors
WHERE    authorId = 1    WHERE   authorId IN K     WHERE   city = “Madison”
```

Figure 3.8 Refresh query examples

```
Presence (Completeness) query:      Consistency query:

SELECT   *                          SELECT  *
FROMV                               FROM    V
WHERE    K = k                      WHERE   Ki IN Ki
```

Figure 3.7 Refresh queries

Next, suppose we have CityList_CsCT (see Section 3.3.1.2). If Madison is not found in CityList_CsCT, the presence query described above is sufficient. Otherwise, we must also refresh all other authors from Madison. If **K** is the set of authors in AuthorCopy that are from Madison, the consistency query in Figure 3.8 is sent to the backend server.

Finally, suppose we have CityList_CpCT (see Section 3.3.1.3). If Madison is found in CityList_CpCT, then besides the consistency query, we must fetch all authors from Madison using the completeness query in Figure 3.8.

Formally, given a cache region U<V, K, k>, let the set of presence control tables of V be $P_1, \ldots, P_n$, with presence control-keys $K_1, \ldots, K_n$. For K_i, i = 1..n, let $\mathbf{K_i}=\Pi_{K_i}\sigma_{K=k}(V)$, the remote queries for U are: 1) the presence query, if U is a presence region; 2) the consistency queries (i = 1..n), if U is a consistency region; and 3) the consistency queries (i = 1..n) (and the completeness query if $U \neq \emptyset$), if U is a completeness region. (The queries are shown in Figure 3.7.)

Lemma 3.6: For any cache region U <V, K, k> in the cache, the results retrieved from the backend server using the refresh queries in Figure 3.7 not only keeps U's cache constraints, but also keeps the presence constraints for the presence regions in V that U overlaps. □

Proof of Lemma 3.6:

This directly follows from the presence, consistency and completeness queries. □

As this example illustrates, when refreshing a cache region, in order to guarantee cache constraints, we may need to refresh additional cache regions; the set of all such "affected" cache regions is defined below.

Definition: (**Affected closure**) The affected closure of a cache region U, denoted as **AC**(U), is defined transitively:

1) AC(U) = {U}
2) AC(U) = AC(U) $\cup$ {U_i | for U_j in AC(U), either U_j overlaps U_i or U_j and U_i are consistency-wise correlated}. □

For convenience, we assume that the calculation of AC(U) always eliminates consistency region U_i, if there exists a completeness region U_j in AC(U), s.t. $U_i = U_j$, since the completeness constraint is stricter (Lemma 3.3). The set of regions in AC(U) is partially ordered by the set containment relationship. From Lemma 3.1 - Lemma 3.3, we only need to maintain the constraints of some “maximal” subset of AC(U). Let **Max**(Ω) denote the set of the maximal elements in the partially ordered set Ω.

Definition: (**Maximal affected closure**) The maximal affected closure of a cache region U, **MaxAC**(U), is obtained by the following two steps: Let Ω = AC(U),

1) Constructing step. Let д, в be the set of all consistency regions and completeness regions in Ω respectively. MaxAC(U) = Max(Ω – д) $\cup$ Max(Ω – в).
2) Cleaning step. Eliminate any consistency region U_i in MaxAC(U) if there exists a completeness region U_j in MaxAC(U), s.t. $U_i \subseteq U_j$. □

Maintenance Rule:

1) We only choose a region to refresh from a white node.
2) When we refresh a region U, we do the following:

   Step 1: Retrieve every region in MaxAC(U) by sending proper remote queries according to its constraint.

   Step 2: Delete the old rows covered by AC(U) or the retrieved tuple set; then insert the retrieved tuple set. □

Theorem 3.1: Assuming the partial order between any two cache regions is constant, then given any region U, if we apply the Maintenance Rule to a cache instance that satisfies all cache constraints, let newTupleSet be the newly retrieved tuple set, Δ = AC(newTupleSet), then

1) Every region other than those in $(\Delta - \Omega)$ observes its cache constraint after the refresh transaction is complete.
2) If $(\Delta - \Omega) = \emptyset$, then after the refresh transaction is complete, all cache constraints are preserved.
3) If $(\Delta - \Omega) = \emptyset$, MaxAC(U) is the minimal set of regions we have to refresh in order to refresh U while maintaining all cache constraints for all cache instances. □

Proof of Theorem 3.1:

Let Ω = AC(U), maxSet=MaxAC(U), newTupleSet be the tuple set retrieved for maxSet.

We first prove 1)

1) For any cache region X <V, K, k> in Ω, let V' be the refreshed instance of V, D be the set of rows for V in newRowSet, $X = \delta_{K=k}$ (V), $X' = \delta_{K=k}$ (V'), and $X'' = \delta_{K=k}$ (D).

   We first prove $X' = X''$. This is obvious from step 2 in the maintenance rule, since all the rows in X are deleted and all the rows in X'' are inserted into V'.

   Case 1: X is in maxSet. Directly from Lemma 3.6.

   Case 2: X is in (Ω–maxSet). Then there is a region Y in maxSet, such that $X \subseteq Y$.

   Case 2.1: If X is a presence region, then directly from Lemma 3.6. Otherwise,

   Case 2.2: Y has an equal or stronger constraint than X. Since Y observes its constraint (from Case 1), it follows from Lemma 3.1, Lemma 3.2 and Lemma 3.3 that so does X.

Case 3: X is not in $\Delta \cup \Omega$. We prove that *X' = X*. This is so because from the maintenance rule, those rows in U are not touched by the refresh transaction.

2) It directly follows from 1).

3) It is obvious if U is the only element in Ω. Otherwise, prove by constructing counterexamples from AuthorCopy. In AuthorCopy, suppose there is a present control table on authorId with authorIds 1 and 2; there are two tuples: t1 = <1, Rose, Female, Madison, WI>, t2 = <2, Mary, Female, Seattle, WA>. Suppose we want to refresh t1 after an update that touched every row in Authors in the master database.

   Prove by contradiction. Suppose there exists X in maxSet that should not be refreshed.

   Case 1: There exists Y in maxSet, such that $X \subseteq Y$. Due to the definition of the maxSet, X must be a completeness region and Y a consistency region.

   In AuthorCopy, suppose it has a completeness region defined on city with value Madison; a consistency region defined on state with value WI. If a new author from Madison has been added in the master database, if we only refresh the consistency region by WI, only t1 will be refreshed, and after refresh, the completeness constraint on Madison is no longer preserved.

   Case 2: There exists a cache region Y in maxSet, s.t. X overlaps with Y. In AuthorCopy, suppose it has two consistency regions on WI and female respectively. If we only refresh the first one, only t1 will be refreshed, and after refresh, the consistency constraint on the latter is no longer preserved. □

The last part of the theorem shows that when a region U is refreshed, every region in MaxAC(U) must be simultaneously refreshed. Otherwise, there is some instance of the cache that satisfies all constraints, yet running the refresh transaction on this state to refresh U will leave the cache in a state violating some constraint. If $(\Delta - \Omega) \neq \emptyset$, multi-trip to the master database is needed in order to maintain all cache constraints. A general maintenance algorithm is sketched in Figure 3.10.

Function retrieve(Ω) retrieves rows from the master database by sending a series of remote queries accordingly for each group in Ω. Procedure apply() as shown in Figure 3.9

```
Algorithm Maintenance
Inputs:    a cache region U from a white node
Outputs:   refreshed U

begin
    Ω ← {U};
    repeat
        Ω ← AC(Ω);
        maxSet ← MaxAC(Ω);
        oldRowSet = $\bigcup_{U_i \in \max Set,} U_i$          // the instance set
        NewRowSet = retrieve(maxSet);
        Δ = AC(NewRowSet);
        if (Δ ⊆ Ω)
            break;
        end if
        Ω = Δ ∪ Ω;
    until (true);
    apply(oldRowSet, newRowSet);
end Maintenance
```

Figure 3.9 Algorithm for updating the cache with newly retrieved tuples

refreshes the cache according to step 2 in the second part of the Maintenance Rule.

Given a region U in a white PMV V, how do we get MaxAC(U)? For an arbitrary cache schema, we need to start with U and add affected regions to it recursively. There are two scenarios that potentially complicate the calculation of MaxAC(U), and could cause it to be very large:

1) For any view V_i, adding a region U_j from V_i results in adding all regions from V_i that overlap with U_j.
2) A circular dependency may exist between two views V_i and V_j, i.e., adding new regions from V_i may result in adding more regions from V_j, which in turn results in adding yet more regions from V_i.

The potentially expensive calculation and the large size of MaxAC(U), and the correspondingly high cost of refreshing the cache motivate the definition of *safe* PMVs in

```
Algorithm  Apply
Inputs:   S - source row set; D - new row set
Output:  a refreshed cache

begin
    for (each view $V_i$ involved)
        Let the set of rows in S that belongs to $V_i$ be $S_i$;
        Let the set of rows in D that belongs to $V_i$ be $D_i$;
        Let dkey = $\Pi_{key}(D_i)$;
        Delete $S_i$ from $V_i$;
        Delete rows in $V_i$ whose keys appear in dkey;
        Insert $D_i$ into $V_i$;
    end for
end Apply;
```

Figure 3.10 Algorithm for refreshing a cache region

Section 3.4.2.

3.4.1.1 Ad-hoc Cache Regions

Although the specified cache constraints are the minimum constraints that the cache must guarantee, sometimes it is desirable for the cache to provide additional “ad-hoc” guarantees. For example, a query workload like E1.1 asks for authors from a set of popular authors and requires them to be mutually consistent. Popularity changes over time. In order to adapt to such workloads, we want the flexibility of grouping and regrouping authors into cache regions on the fly. For this purpose, we allow the cache to group regions into “ad-hoc” cache regions.

Definition: (**Ad-hoc region)** An ad-hoc cache region consists of a union of one or more regions (which might be from different views) that are mutually consistent. □

Such “ad-hoc” consistency information is made known to the query processor by associating the region id of the ad-hoc region with each region it contains.

3.4.1.2 Keeping Track of Currency

In order to judge if cached data is current enough for a given query, we need to keep track of its currency. It is straightforward and we discuss it only briefly. Chapter 4 used a “push” model for cache maintenance, and relied on a heartbeat mechanism for this purpose. To track currency when using the pull model, we keep a timestamp for every cache region. When a cache region is refreshed, we also retrieve and record the transaction timestamp of the refresh query. Assuming that a transaction timestamp is unique, in implementation we simply use the timestamp as region id. Thus, if the timestamp for a cache region is *T* and the current time is *t*, since all updates until *T* are reflected in the result of the refresh query, the region is from a database snapshot no older than *t* – *T*.

3.4.2 Safe Views and Efficient Pulling

We now introduce the concept of *safe* views, motivated by the potentially high refresh cost of pull-based maintenance for unrestricted cache schemas.

Definition: (Safe PMV) A partially materialized view V is safe if the two following conditions hold for every instance of the cache that satisfies all integrity constraints:

1) For any pair of regions in V, either they don't overlap or one is contained in the other.
2) If V is gray, let X denote the set of presence regions in V. X is a partitioning of V and no pair of regions in X is contained in any one region defined on V. □

Intuitively, Condition 1 is to avoid unexpected refreshing because of overlapping regions in V; Condition 2 is to avoid unexpected refreshing because of consistency correlation across nodes in the cache schema.

Lemma 3.7: For a safe white PMV V that doesn't have any children, given any cache region U in V, the partially ordered set AC(U) is a tree. □

Proof of Lemma 3.7:

(By contradiction)

Suppose there is a group X in AC(U), such that X has two parents Y and Z. Then $Y \cap Z \neq \emptyset$. From the safe definition, either $Y \subseteq Z$, or $Z \subseteq Y$. Therefore they cannot both be X's parents. □

Since AC(U) on V has a regular structure, the maximal element can be find efficiently.

Theorem 3.2: Consider a white PMV V, and let κ denote V and all its gray descendants. If all nodes in κ are safe, whenever any region U defined on V is to be refreshed:

1) AC(U) can be calculated top-down in one pass.
2) Given the partially ordered set AC(U) on V, the calculation of MaxAC(U) on V can be done in one pass. □

Proof of Theorem 3.2:

1) For any safe gray node V', given the subset of PCK values **K** that is in AC(U) from its parent, we need to put in AC(U) the set of cache regions Δ determined by **K** in V'. Δ is the exact set of cache regions in V' that need to be put into AC(U), because from the definition of a safe view, Δ doesn't overlap or is contained by any consistent or completeness region defined on V', nor does it overlap or is contained by the rest of the present CRs in V'. Further, adding Δ to AC(U)

doesn't result in adding additional cache regions from its parent, because of the first condition of the definition of safe.

2) From 1), the descendents of V don't affect AC(U) on V. Thus, let Ω = AC(U), from Lemma 3.7, Ω is a tree. Let Γ be empty, we check the tree recursively top down from the root, let it be Y. If a node X is a completeness region, then we add it to Γ; otherwise, we do the checking on each child of X. If Y is not in Γ, add it to Γ.

   We prove that Γ = MaxAC(U). If Y is a complete or a presence region, we are done. Otherwise, let д, в be the set of all consistency regions and completeness regions in Ω respectively. {Y} = Max (Ω – в), since it is the root of the tree. Now we prove Γ – {Y} = Max(Ω – д) by contradiction. Suppose there is a completeness region Z in Ω, such that Γ – {Y} doesn't cover Z. Then Z doesn't have any ancestor that is a completeness region. Then from the algorithm, Z must be visited and put into Γ – {Y}, contradicting the assumption.

   Further, the cleaning step doesn't eliminate Y, since it is the root. Thus Γ = MaxAC(U). □

Rule 1:	A cache graph is a DAG.
Rule 2:	Only white nodes can have independent completeness or consistency control tables.
Rule 3:	A view with more than one parent must be a white circle.
Rule 4:	If a view has the shared-row problem according to Lemma 5.2, then it cannot be gray.
Rule 5:	A view cannot have incompatible control tables.

Figure 3.11 Cache schema design rules

3.5 Design Issues for Caches

In this section, we investigate conditions that lead to unsafe cached views and propose appropriate restrictions on allowable cache constraints. In particular, we develop three additional rules to guide cache schema design, and show that Rules 1-5 are a necessary and sufficient condition for (all views in) the cache to be safe.

3.5.1 Shared-Row Problem

Let's take a closer look at the AuthorCopy and BookCopy example defined in Section 3.3. Suppose a book can have multiple authors. If BookCopy is a rectangle, since co-authoring is allowed, a book in BookCopy may correspond to more than one control-key (authorId) value, and thus belong to more than one cache region. To reason about such situations, we introduce cache-instance DAGs.

Definition: (Cache instance DAG) Given an instance of a cache DAG <**W**, **E**>, we construct its cache instance DAG as follows: make each row in each node of **W** a node; and for each edge $V_i \xrightarrow{K_{i,j}, K_{i,j}'} V_j$ in **E**, for each pair of rows s in V_i and t in V_j, if $s.K_{i,j} = t.K_{i,j}$' then add an edge s → t. □

Definition: (Shared-row problem) For a cache DAG <**W**, **E**>, a view V in **W** has the shared-row problem if there is an instance DAG s.t. a row in V has more than one parent. □

There are two cases where a view V has the shared-row problem. In the first case (Lemma 3.8), we can only eliminate the potential overlap of regions in V defined by different presence control tables if V is view-level consistent. Considering the second condition in the definition of safe, we have Rule 3 in Figure 3.11. For the second case (Lemma 3.9) we enforce Rule 4 in Figure 3.11.

Lemma 3.8: Given a cache schema <**W**, **E**>, view V in **W** has the shared-row problem if V has more than one parent. □

Proof of Lemma 3.8:

> (By constructing an instance DAG). Suppose V has two PCTs T1 and T2 on attributes A and B respectively. Suppose values a1 and b1 are in T1 and T2 respectively. For a row t in V, if t.A = a1, t.B = b1, then t has two parents: a1 and b1. Thus V has the shared-row problem. □

Lemma 3.9: Given a cache schema <**W**, **E**>, for any view V, let the parent of V be V_1. V has the shared-row problem iff the presence key K in V_1 for V is not a key in V_1. □

Proof of Lemma 3.9:

(sufficiency) Since K is not a key for V_1, there exists an instance of V_1, such that there are two rows t1 and t2 in V_1, t1.K = t2.K. Then for a row t in V, s.t. t.K = t1.K, both t1 and t2 are t's parents.

(necessity) Because V has the shared-row problem, there is an instance of V, such that a row t in V has two parents, t1 and t2 in V_1. Since t1.K = t2.K= t.K, K is not a key for V_1. □

3.5.2 Control-Table Hierarchy

For a white view V in the cache, if it has consistency or completeness control tables beyond those implicit in the Presence Assumption, then it may have overlapping regions. In our running example, suppose BookCopy is a white rectangle; an author may have more than one publisher. If there is a consistency control table on publisherId, then BookCopy may have overlapping regions. As an example, Alice has books 1 and 2, Bob has book 3, and while books 1 and 3 are published by publisher A, book 2 is published by publisher B. If publisher A is in the consistency control table for BookCopy, then we have two overlapping regions: {book 1, book 2} by Alice, and {book 1, book 3} by publisher A.

Definition: (Compatible control tables) For a view V in the cache, let the presence controlled-key of V be K_0, and let the set of its consistency and completeness control-keys be **K**.

1) For any pair K_1 and K_2 in **K**, we say that K_1 and K_2 are compatible iff FD $K_1 \rightarrow K_2$ or FD $K_2 \rightarrow K_1$.
2) We say **K** is compatible iff the elements in **K** are pair-wise compatible, and for any K in **K**, FD $K \rightarrow K_0$. □

Rule 5 is stated in Figure 3.11. We require that a new cache constraint can only be created in the system if its addition does not violate Rules 1-5.

Theorem 3.3: Given a cache schema <**W**, **E**>, if it satisfies rules 1-5, then every view in **W** is safe. Conversely, if the schema violates one of these rules, there is an instance of the cache satisfying all specified integrity constraints in which some view is unsafe. □

Proof of Theorem 3.3:

(Sufficiency) by contradiction. Suppose there exists a PMV V that is not safe. There are two cases:

Case 1: There exists a pair of cache regions U1 and U2 in V, s.t. U1 and U2 overlap. This violates Rule 5.

Case 2: V is grey. Let Ω denote the set of cache regions in V defined by its presence control-key values. Again, there are two cases:

Case 2.1: There are U1 and U2 in Ω, such that U1 and U2 overlap.

This implies that V has shared-row problem. Then it violates rule 3 or 4.

Case 2.2: There are U1 and U2 in Ω, and U3 in V, such that U1 and U2 are contained in U3.

This implies that V has its own consistency control-tables, which violates rule 2.

(Necessity) We use variations of the cache schema in Figure 3.11 as counter examples in a proof by contradiction.

Case 1: Rule 1 is violated. Then <W, E> violates the definition of cache schema.

Case 2: Rule 2 is violated.

Suppose BookCopy is required to be consistent by type; author a1 has books b1 and b2; a2 has a book b3; and b1, b2, b3 are all of type paperback. Then BookCopy is not safe because cache regions {b1, b2} (by a1), {b3} (by a2) are contained in the one defined by paperback type.

Case 3: Rule 3 is violated.

Suppose ReviewCopy is a rectangle or gray. If it is a rectangle, suppose book b1 has two reviews r1, and r2, from reviewers x and y, respectively; x wrote reviews r1 and r3. Since cache regions {r1, r2} (by b1) and {r1, r3} (by x) overlap, ReviewCopy is not safe.

Next, if ReviewCopy is a circle, suppose author a1 has books b1 and b2; author a2 has a book b3; books b2, b3 have reviews r2, r3, respectively. Since cache regions {b1, b2} (by a1) and {b2, b3} (by correlation with ReviewCopy), BookCopy is not safe.

Case 4: Rule 4 is violated.

Suppose a book can have multiple authors and BookCopy is gray. Suppose AuthorCopy is consistent by city; author a1 has books b1 and b2; author a2 has books b1 and b3; author a1 and a3 are from WI, a2 is from WA.

First, suppose BookCopy is a rectangle. Since cache regions {b1, b2} (by a1), {b1, b3} (by a2) overlap, BookCopy is not safe.

Second, suppose BookCopy is a circle. Since cache regions {a1, a3} (by WI), and {a1, a2} (by consistency correlation with BookCopy) overlap, AuthorCopy is not safe.

Case 5: Rule 5 is violated.

Suppose ReviewerCopy is required to be consistent both by gender and by city; reviewers x and y are from WI, z is from WA; x and z are male, while y is female. Since cache regions: {x, y} (by WI), {x, z} (by male) overlap, ReviewCopy is not safe. □

Chapter 4

Enforcing Data Quality Constraints for View-level Granularity

How can we efficiently ensure that a query result meets the stated C&C requirements? This chapter answers this question for a simplified case where 1) only fully materialized views are allowed in the cache schema; 2) all the rows in a view are mutually consistent. We explain how C&C constraints are enforced in MTCache, a prototype mid-tier database cache, built on the Microsoft SQL Server codebase, including how constraints and replica update policies are elegantly integrated into the cost-based query optimizer. Consistency constraints are enforced at compile time while currency constraints are enforced at run time with dynamic plans that check the currency of each local replica before use and select sub-plans accordingly. This approach makes optimal use of the cache DBMS while at the same time guaranteeing that applications always get data with sufficient quality for their purpose.

The rest of the chapter is organized as follows. Section 4.1 gives an overview of the MTCache framework. Section 4.2 describes mechanisms to keep track of local data quality. We develop techniques to enforce C&C during query processing in Section 4.3 and report analytical and experimental results in Section 4.4.

4.1 MTCache Framework Overview

We have implemented support for explicit C&C constraints as part of our prototype mid-tier database cache, MTCache, which is based on the following approach:

1) A shadow database is created on the cache DBMS, containing the same tables as the back-end database, including constraints, indexes, views, and permissions, but with all tables empty. However, the statistics maintained on the shadow tables, indexes and materialized views reflect the data on the back-end server rather than the cache.
2) What data to cache is defined by creating materialized views on the cache DBMS. These materialized views may be selections and projections of tables or materialized views on the back-end server.
3) The materialized views on the cache DBMS are kept up to date by SQL Server's transactional replication [Ise01, Hen04]. When a view is created, a matching replication subscription is automatically created and the view is populated.
4) All queries are submitted to the cache DBMS, whose optimizer decides whether to compute a query locally, remotely, or partly locally and partly remotely. Optimization is entirely cost based.
5) All inserts, deletes and updates are submitted to the cache DBMS, which then transparently forwards them to the back-end server.

We have extended this cache prototype to support queries with C&C constraints. We keep track of which materialized views are mutually consistent (reflect the same database snapshot) and how current their data is. We extended the optimizer to select the best plan taking into account the query’s C&C constraints and the status of applicable local materialized views. In contrast with traditional plans, the generated plan includes run-time checking of the currency of each local view used. Depending on the outcome of this check, the plan switches between using the local view and submitting a remote query. The result returned to the user is thus guaranteed to satisfy the query’s consistency and currency constraints.

Our prototype currently only supports table-level consistency and does not allow C&C constraints with grouping columns, such as described by the phrase “BY B.isbn” in E4. They would have no effect in any case because all rows within a local view are always mutually consistent (they are updated by transactional replication).

We rely on the SwitchUnion operator in SQL Server. This operator has N+1 input expressions. When opening the operator, one of the first N inputs is selected and all rows are taken from that input; the other N–1 inputs are not touched. Which input is selected is determined by the last input expression, here called the selector expression. The selector must be a scalar expression returning a number in the range 0 to N–1. The selector expression is first evaluated and the number returned determines which one of the first N inputs to use. We use a SwitchUnion operator to transparently switch between retrieving data from a local view and retrieving it with a query to the back-end server. The selector expression checks whether the view is sufficiently up-to-date to satisfy the query’s currency constraint.

4.2 Cache Regions

To keep track of which materialized views on the cache DBMS are mutually consistent and how current they are, we group them into logical cache regions. The maintenance mechanisms and policies must guarantee that all views within the same region are mutually consistent at all times.

Our prototype relies on SQL Server's transactional replication feature [Ise01, Hen04] to propagate updates from the back-end database to the cache. Updates are propagated by distribution agents. (A distribution agent is a process that wakes up regularly and checks for work to do.) A local view always uses the same agent but an agent may be responsible for multiple views. The agent applies updates to its target views one transaction at a time, in commit order. This means that all cached views that are updated by the same agent are mutually consistent and always reflect a committed state. Hence, all views using the same distribution agent form a cache region.

Our current prototype is somewhat simplified and does not implement cache regions as separate database objects. Instead, we added three columns to the catalog data describing views: cid, update_interval, update_delay. Cid is the id of the cache region to which this view belongs. Update_interval is how often the agent propagates updates to this region. Update_delay is the delay for an update to be propagated to the front-end, i.e., the minimal currency this region can guarantee. Update_delay and update_interval can be estimates because they are used only for cost estimation during optimization.

Our mechanism for tracking data currency is based on the idea of a *heartbeat*. We have a global heartbeat table at the back-end, containing one row for each cache region. The table has two columns: a cache region id and a timestamp. At regular intervals, say every 2 seconds, the region's heart beats, that is, the timestamp column of the region's row is set to the current timestamp by a stored procedure. (Another possible design uses a heartbeat table with a single row that is common to all cache regions, but this precludes having different heartbeat rates for different regions.)

Each cache region replicates its row from the heartbeat table into a local heartbeat table for the region. The agent corresponding to the cache region wakes up at regular intervals and propagates all changes, including updates to the heartbeat table. The timestamp value in the local heartbeat table gives us a bound on the staleness of the data in that region. Suppose the timestamp value found in the region's local heartbeat table is T and the current time is t. Because we are using transactional replication, we know that all updates up to time T have been propagated and hence reflect a database snapshot no older than $t – T$.

4.3 Implementation

A traditional distributed query optimizer decides whether to use local data based on data availability and estimated cost. In our setting, it must also take into account local data properties (presence, consistency, completeness and currency), thereby guaranteeing that the result it produces satisfies the data quality requirements specified in a query. Our approach depends on two key techniques of Microsoft's SQL Server optimizer: 1) required and delivered plan properties; and 2) the SwitchUnion operator. We translate C&C requirements into a normalized form (Section 4.3.1). We then employ the required and delivered plan properties framework to perform consistency checking during query compilation (Section 4.3.2), and use the SwitchUnion operator to enforce currency checking at query execution time (Section 4.3.3). Further, a cost estimator for the SwitchUnion operator is developed (Section 4.3.4), giving the optimizer the freedom to choose the best plan based solely on cost.

4.3.1 Normalizing C&C Constraints

We extended the SQL parser to also parse currency clauses. The information is captured, table/view names resolved and each clause converted into a C&C constraint of the form below.

Definition: (**Currency and consistency constraint**) A C&C constraint C is a set of tuples, C = {<b_1**, $\mathbf{S_1}$**>,…, {<b_n**, $\mathbf{S_n}$**>}, where each $\mathbf{S_i}$ is a set of input operands (table or view instances) and b_i is a currency bound specifying the maximum acceptable staleness of the input operands in $\mathbf{S_i}$. □

C&C constraints are sets (of tuples), so constraints from different clauses can be combined by taking their union. (After name resolution, all input operands reference unique table or view inputs; the block structure of the originating expression affects only name resolution.) We union together all constraints from the individual clauses into a single constraint, and convert it to a normalized form with no redundant or contradictory requirements.

Definition: (**Normalized C&C constraint**) A C&C constraint C = {<b_1**, S_1**>,…, {<b_n**, S_n**>} is in normalized form if all input operands (in the sets **S_i**) are base tables and the input operand sets **S_1**,…, **S_n** are all non-overlapping. □

```
Algorithm NormalizeConstraint
Inputs: A C&C constraint C
Outputs: A normalized C&C constraint C'

begin
    C' = C;
    // Step 1: Eliminate references to views
    for (each tuple <c, S> in C')
        while (exists v $\in$ S such that v is a view)
          replace v in S by the input operands of the view expression;
        end while
    end for

    // Step 2: Combine overlapping tuples
    while (exist $p_1$= <$c_1$, $S_1$> and $p_2$ = <$c_2$, $S_2$> in C' s.t. $S_1 \cap S_2 \neq \emptyset$)
          p = <min($c_1$, $c_2$), ($S_1 \cup S_2$) > ;
          delete $p_1$ and $p_2$ from C';
          add p to C'
    end while

    // Step 3: Add default requirement
    S = all input operands of the query that are not included in C';
    if (S $\neq \emptyset$)
        add the tuple <0, S> to C';
end NormalizeConstraint
```

Figure 4.1 Algorithm for C&C constraint normalization

The first condition simply ensures that the input sets all reference actual input operands of the query (and not views that have disappeared as a result of view expansion). The second condition eliminates redundancy and simplifies checking.

The algorithm that transforms a set of constraints into normalized form is shown in Figure 4.1. At step 1, the algorithm recursively expands all references to views into references to base tables. At step 2, it repeatedly merges all tuples that have one or more input operands in common. The bound for the new tuple is the minimum of the bounds of the two input tuples. Input operands referenced in a tuple must all be from the same database snapshot. It immediately follows that if two different tuples have any input operands in common, they must all be from the same snapshot, and the snapshot must satisfy the tighter of the two bounds. The merge step continues until all tuples are disjoint. Step 3 simply adds a default requirement on all input operands not yet covered. We chose as the default the tightest requirements, namely, that the input operands must be mutually consistent and from the latest snapshots, i.e., fetched from the back-end database. This tight default has the effect that queries without an explicit currency clause will be sent to the back-end server and their result will reflect the latest snapshot. In other words, queries without a currency clause retain their traditional semantics.

4.3.2 Compile-time Consistency Checking

SQL Server uses a transformation-based optimizer, i.e., the optimizer generates rewritings by applying local transformation rules on subexpressions of the query. Applying a rule produces substitute expressions that are equivalent to the original expression. Operators are of two types: logical and physical. A logical operator specifies what algebraic operation to perform, for example, a join, but not what algorithm to use. A physical operator also specifies the algorithm, for example, a hash join or merge join. Conceptually, optimization proceeds in two phases: an exploration phase and an optimization phase. The exploration phase generates new logical expressions, that is, algebraic alternatives. The optimization phase recursively finds the best physical plan, that is, the best way of evaluating the query. Physical plans are built bottom-up, producing plans for larger and larger sub-expressions.

Required and delivered (physical) plan properties play a very important role during optimization. There are many plan properties but we'll illustrate the idea with the sort property. A merge join operator requires that its inputs be sorted on the join columns. To ensure this, the merge join passes down to its inputs a required sort property (a list of sort columns and associated sort order). In essence, the merge join is saying: "Find me the cheapest plan for this input that produces a result sorted on these columns." Every physical plan includes a delivered sort property that specifies if the result will be sorted and, if so, on what columns and in what order. Any plan whose delivered properties do not satisfy the required properties is discarded. Among the qualifying plans, the one with the estimated lowest cost is selected.

To integrate consistency checking into the optimizer we must specify and implement required consistency properties, delivered consistency properties, and rules for deciding whether a delivered consistency property satisfies a required consistency property.

4.3.2.1 Required Consistency Plan Property

A query's required consistency property consists precisely of the normalized consistency constraint described above that is computed from the query's currency clauses. The constraint is attached as a required plan property to the root of the query. A pointer to this property is inherited recursively by its children.

4.3.2.2 Delivered Consistency Plan Property

A delivered consistency property consists of a set of tuples {<R_i, $\mathbf{S_i}$>} where R_i is the id of a cache region and $\mathbf{S_i}$ is a set of input operands, namely, the input operands of the current expression that belong to region R_i.

Delivered plan properties are computed bottom-up. Each physical operator (select, hash join, merge join, etc.) computes what plan properties it delivers given the properties of its inputs. We can divide the physical operators into four categories, each using a specific algorithm to compute the delivered consistency property. The algorithm is shown in Figure 4.2.

The leaves of a plan tree are table or index scan operators, possibly with a range predicate. If the input operand is a base table (or an index on a base table), we simply return the id of the table and the id of its cache region. Consistency properties always refer to base tables. Hence, a scan of a materialized view returns the ids of the view's input tables, not the id of the view.

```
Algorithm DrvdPropertyCalc
Inputs: op – an operator
Outputs: CPd

Begin
    switch (the type of op)
    // case 1: table or index scan operator, possibly with a range predicate
    case leaf operator:
        CPd = {<R, S>}, which is directly obtained from the table, view or index
            referred to by op;
        break;

    // case 2: operators with only one relational operand
    case Single operand  operator:
        CPd = the property of the relational operand of op;
        break;

    // case 3: join operators with a set of operands
    case join operator:
        Let {$CPd_i$} be the set of properties of op's relational operands;
        CPd = $\cup$ $CPd_i$;
        repeat
            Merge <$R_1$, $S_1$ >and <$R_2$, $S_2$, $\Omega_2$> in CPd by <R, S> if
              R1 = R2, where R = $R_1$, $S = S_1 \cup S_2$.
        end repeat
        break;

    // case 4: SwitchUnion operator
    case SWU operator:
        Let {$CPd_i$} be the set of properties of op's relational operands, i=1..n;
            CPd = $CPd_1$;        // initialization
            for each $CPd_i$ , where i = 2..n
                CPdTemp = Ø; //initialization
                for each <R, S> in $CPd_i$
                        CPdTemp  $\cup$ = IntersectByPhrase(CPd, <R, S>);
                end for
                CPd = CPdTemp;
            end for
        break;
end DrvdPropertyCalc
```

Figure 4.2 Algorithm for calculating derived C&C property

All operators with a single relational input such as filter, project, aggregate, and sort do not affect the delivered consistency property and simply copy the property from its relational input.

Join operators combine two or more input streams into a single output stream. We compute the consistency property of the output from the consistency properties of the two (relational) children. If the two children have no inputs from the same cache region, the output property is simply the union of the two child properties. If they have two tuples with the same region id, the input sets of the two tuples are merged.

```
Algorithm IntersectByPhrase
Inputs:   $CPd_1$ – derived C&C property,
          <R, S> – a phrase of derived C&C property
Outputs:  CPd – intersected C&C property

begin
    CPd = $\emptyset$;
    // check all phrases in $CPd_1$ that intersect <R, S>
    for each phrase <$R_i$, $S_i$> in $CPd_1$, where $S \cap S_i \neq \emptyset$
        // decide the correct region id
        if (R == $R_i$)
            $R_2$ = R;
        else
        // local is weaker than remote, take the weaker guarantee
        if one of the region is remote, w/o loss of generality, suppose $R_1$
            $R_2$ = R;
        else   //both local, then $R_2$ is undecided
            $R_2$ = UNDECIDED;
        end if
        CPd $\cup$ = {<$R_2$, ($S_1 \cap S_2$)>};
    end for
end IntersectByPhrase
```

Figure 4.3 Algorithm for calculating intersection of C&C property

A SwitchUnion operator has multiple input streams but it does not combine them in any way; it simply selects one of the streams. So how do we derive the delivered consistency of a SwitchUnion operator? The basic observation is that we can only guarantee that two input operands are consistent if they are consistent in all children (because any one of the children may be chosen). The algorithm repeatedly calculates the common property guaranteed by two inputs. A subroutine algorithm in Figure 4.2 calculates the common property from a single phrase and a C&C property (i.e., a set of phrases).

4.3.2.3 Satisfaction Rules

Plans are built bottom-up, one operator at a time. As soon as a new root operator is added to a plan, the optimizer checks whether the delivered plan properties satisfy the required plan properties. If not, the plan, i.e., the new root operator, is discarded. We include the new consistency property in this framework.

Our consistency model does not allow two columns from the same input table T to originate from different snapshots. It is possible to generate a plan that produces a result with this behavior. Suppose we have two (local) projection views of T that belong to different cache regions, say R1 and R2, and cover different subsets of columns from T. A query that requires columns from both views could then be computed by joining the two views. The delivered consistency property for this plan would be {<R1, T>, <R2, T>}, which conflicts with our current consistency model. Here is a more formal definition.

Conflicting consistency property: A delivered consistency property CPd is conflicting if there exist two tuples <R_i, $\mathbf{S}_i$> and <R_j, $\mathbf{S}_j$> in CPd such that $\mathbf{S}_i \cap \mathbf{S}_j \neq \emptyset$ and $R_i \neq R_j$.

A consistency constraint specifies that certain input operands must belong to the same region (but not which region). We can verify that a complete plan satisfies the constraint by checking that each required consistency group is fully contained in some delivered consistency group. The following rule is based on this observation.

Consistency satisfaction rule: A delivered consistency property CPd satisfies a required consistency constraint CCr if and only if CPd is not conflicting and, for each tuple <B_r, $\mathbf{S}_r$> in CCr, there exists a tuple <R_d, $\mathbf{S}_d$> in CPd such that $\mathbf{S}_r$ is a subset of $\mathbf{S}_d$.

While easy to understand, this rule can only be applied to complete plans because a partial plan may not include all input operands covered by the required consistency property. We need a rule that allows us to discard partial plans that do not satisfy the required consistency property as soon as possible. We use the following rule on partial plans to detect violations early.

Consistency violation rule: A delivered consistency property CPd violates a required consistency constraint CCr if (1) CPd is conflicting or (2) there exists a tuple <R_d, $\mathbf{S}_d$> in CPd that intersects more than one consistency class in CCr, that is, there exist two tuples <$B1_r$, $\mathbf{S1}_r$> and <$B2_r$, $\mathbf{S2}_r$> in CCr such that $\mathbf{S}_d \cap \mathbf{S1}_r \neq \emptyset$ and $\mathbf{S}_d \cap \mathbf{S1}_r \neq \emptyset$.

We also added a simple optimization to the implementation. If the required currency bound is less than the minimum delay that the cache region can guarantee, we know at compile time that data from the region cannot be used to answer the query. In that case, the plan is immediately discarded.

4.3.3 Run-time Currency Checking

Consistency constraints can be enforced during optimization, but currency constraints must be enforced during query execution. The optimizer must thus produce plans that check whether a local view is sufficiently up-to-date and switch between using the local view and retrieving the data from the back-end server. For this purpose, we use the SwitchUnion operator described earlier.

Recall that all local data is defined by materialized views. Logical plans making use of a local view are always created through view matching, that is, the view matching algorithm finds an expression that can be computed from a local view and that produces a new substitute for the original expression, but exploiting the view. More details about the view matching algorithm can be found in [GL01].

Consider a (logical) expression E and a matching view V from which E can be computed. If there are no currency constraints on the input tables of E, view matching produces a "normal" substitute consisting of, at most, a select, a project and a group-by on top of V. If there is a currency constraint, view matching produces a substitute consisting of a SwitchUnion on top, with a selector expression that checks whether V satisfies the currency constraint, as shown in Figure 4.4. The SwitchUnion has two input expressions: a local branch and a remote branch. The local branch is the "normal" substitute mentioned earlier and the remote plan consists of a remote SQL query created from the original expression E. If the selector expression, which we call the currency guard, evaluates to true, the local branch is chosen, otherwise the remote branch is chosen. SwitchUnion operators are

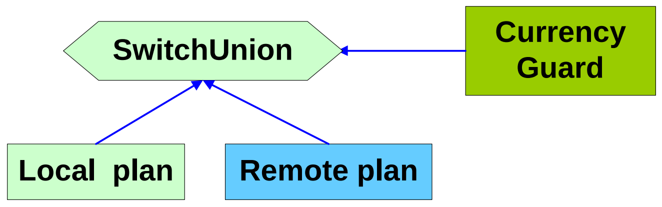


Figure 4.4 Substitute with a SwitchUnion and a currency guard

generated at the leaf-level but they can always be propagated upwards and merged with adjacent SwitchUnion operators. However, these and other optimizations involving SwitchUnion are left as future work.

As mentioned earlier, we track a region's data currency using a heartbeat mechanism. The currency guard for a view in region R is an expression equivalent to the following SQL predicate:

```
EXISTS ( SELECT     1       FROM Heartbeat_R
         WHERE      TimeStamp > getdate() – B )
```

where Heartbeat_R is the local heartbeat table for region R, and B is the applicable currency bound from the query.

The above explanation deliberately ignores the fact that clocks on different servers may not be synchronized. This complicates the implementation but is not essential to understanding the approach.

4.3.4 Cost Estimation

For a SwitchUnion with a currency guard we estimate the cost as

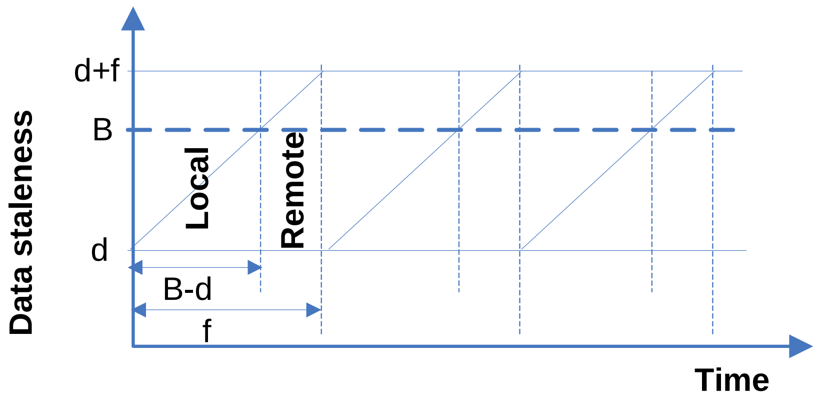


Figure 4.5 Synchronization cycle and data currency

$$c = p * c_{local} + (1 - p) * c_{remote} + c_{cg}$$

where p is the probability that the local branch is executed, c_{local} is the cost of executing the local branch, c_{remote} the cost of executing the remote branch, and c_{cg} the cost of the currency guard. This approach is similar to that of [CHS99, DR99].

The cost estimates for the inputs are computed in the normal way but we need some way to estimate p. We'll show how to estimate p assuming that updates are propagated periodically, the propagation interval is a multiple of the heartbeat interval, their timing is aligned, and query start time is uniformly distributed.

Denote the update propagation interval by f and the propagation delay as d. The currency of the data in the local view goes through a cycle illustrated in Figure 4.5. Immediately after propagation, the local data is no more than d out of date (the time it took to deliver the data). The currency of the data then increases linearly with time to d+f when the next propagation event takes place and the currency drops to d.

Suppose the query specifies a currency bound of *B*. The case when $d < B < d+f$ is illustrated in the figure. The execution of the query is equally likely to start at any point during a propagation cycle. If it starts somewhere in the interval marked “Local”, the local view satisfies the currency constraint and the local branch is chosen. The length of this interval is $B - d$ and the total length of the cycle is *f* so the probability that the local branch will be chosen is $(B - d)/f$.

There are two other cases to consider. If $B<d$, the local branch is never chosen because the local data is never sufficiently fresh so $p=0$. On the other hand, if $B > d+f$, the local branch is always chosen because the local data is always sufficiently fresh so $p=1$. In summary, here is the formula used for estimating p:

$$p = \begin{cases} 0 & \text{if } B - d \leq 0 \\ (B-d)/f & \text{if } 0 < B - d \leq f \\ 1 & \text{if } B - d > f \end{cases} \qquad (1)$$

The special case when updates are propagated continuously is correctly modeled by setting $f = 0$. Then if $B > d$, we have $p = 1$; otherwise, $p = 0$.

4.4 Analysis and Experiments

This section reports analytical and experimental results using our prototype. We show how the choice of query plan is affected as the query’s C&C constraint changes. We also analyze the overhead of plans with currency guards.

4.4.1 Experimental Setup

For the experiments we used a single cache DBMS and a back-end server. The back-end server hosted a TPCD database with scale factor 1.0 (about 1GB). The experiments reported here used only the Customer and Orders tables, which contained 150,000 and 1,500,000 rows, respectively. The Customer table was clustered on its primary key, c_custkey, and had a secondary index on c_acctbal. The Orders table was clustered on its primary key, (o_custkey, o_orderkey).

The cache DBMS had a shadow TPCD database with empty tables but with statistics reflecting the database on the back-end server. There were two local views:

```
cust_prj(c_custkey, c_name, c_nationkey, c_acctbal)
orders_prj(o_custkey, o_orderkey, o_totalprice)
```

Which are projections of the Customer and the Orders tables, respectively. Cust_prj had a clustered index on the primary key c_custkey and orders_prj had a clustered index on (o_custkey, o_orderkey). They had no secondary indexes. The views were in different cache regions and, hence, not guaranteed to be consistent. The propagation intervals and delays are shown in Table 4.1.

	cid	**interval**	**delay**	**views**
CR1	1	15	5	cust_prj
CR2	2	10	5	orders_prj

Table 4.1 Cache region settings

4.4.2 Workload Distribution (Analytical Model)

Everything else being equal, one would expect that when currency requirements are relaxed further, more queries can be computed using local data and hence more of the workload is shifted to the cache DBMS. We will show how the workload shifts when the currency bound *B* is gradually increased in Q7 (Figure 4.7).

The query plan for Q7 uses either the view cust_prj or a remote query. If the query is executed repeatedly, how often can we expect it to run locally and how does this depend on the currency bound *B*?

```
S1: SELECT   c_custkey, c_name, o_orderkey, o_totalprice
    FROM     customer, orders
    WHERE    c_custkey = o_custkey [AND c_custkey<$K]
             [CURRENCY clause]
S2: SELECT   c_custkey, c_name  FROM customer
    WHERE    c_acctbal between $A and $B
    CURRENCY BOUND 10 on (customer)
```

Figure 4.6 Query schemas used for experiments

Q1: $K = 500	P1
Q2: No range predicate; no currency clause	P2
Q3: $K = 500; BOUND 10 ON (customer, orders)	P1
Q4: $K = 500; BOUND 2 ON (customer), BOUND 20 on (orders)	P4
Q5: $K = 500; BOUND 10 ON (customer), BOUND 20 on (orders)	P5
Q6: $A = 100; $B = 105	P1
Q7: $A = 100; $B = 110	P3

Figure 4.7 Query variants used for experiments

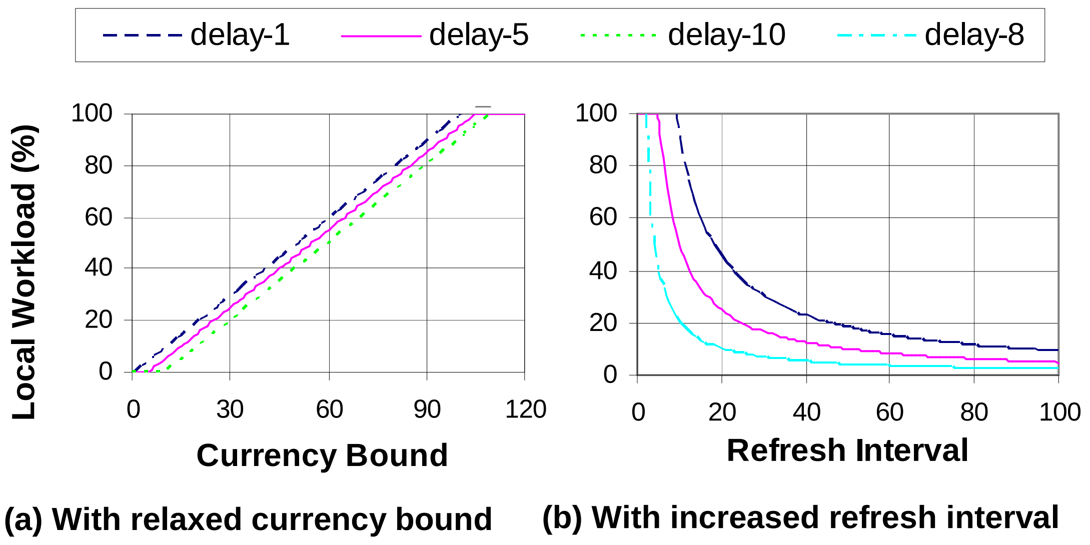


Figure 4.8 Local workload at the cache

Assuming query arrivals follow a Poisson process, we plotted function (1) from Section 4.3.4 in Figure 4.8. In Figure 4.8 (a) it is plotted as a function of the currency bound *B* for *f* = 100 and *d* = 1, 5, 10, respectively. When the currency bound is less than the delay, the query is never executed locally. As the currency bound is relaxed, the fraction of queries executed locally increases linearly until it reaches 100%. This level is reached when $B = d+f$, i.e., when it exceeds the maximal currency of local data. When the delay increases, the curve just shifts to the right.

Figure 4.8 (b) shows the effects of varying the refresh interval. We fixed *B* = 10 and chose *d* = 1, 5, 8, respectively. When the refresh interval is sufficiently small, that is, $f \leq B - d$, the query can always be computed locally. When the refresh interval is increased, more of the workload shifts to the back-end. The effect is much more significant at the beginning and slows down later.

4.4.3 Query Optimization Experiments

We have fully integrated currency and consistency considerations into the cost-based optimizer. The first set of experiments demonstrate how the optimizer's choice of plan is affected by a query's currency and consistency requirements, available local views, their indexes and how frequently they are refreshed.

We used different variants of the query schemas in Figure 4.6, obtained by varying the parameter $K and the currency clause in S1 for Q1 to Q5; $A and $B in S2 for the rest. The parameter values used and the logical plans generated are shown in and Figure 4.7 and Figure 4.9, respectively. The rightmost column in Figure 4.7 indicates which plan was chosen for each query.

If we do not include a currency clause in the query, the default requirements apply: all inputs mutually consistent and currency bound equal to zero. Q1 and Q2 do not include a currency clause. Since local data can never satisfy the currency requirement, remote queries were generated. Because of the highly selective predicate in Q1, the optimizer selected plan 1, which sends the whole query to the back-end. For Q2, plan 2 was selected, which contains a local join and two remote queries, each fetching a base table. In this case, it is better to compute the join locally because the join result is significantly larger (72 MB) than the sum of the two sources (42 MB). Customers have 10 orders on average so the information for a customer is repeated 10 times in the join result.

A remote plan (plan 1) is also generated for Q3 but for a different reason. The applicable local views cust_prj and orders_prj satisfy the currency bounds but not the consistency requirement because they are in different cache regions. In Q4 we relaxed the consistency requirement between Customer and Orders and changed their currency bounds (lower on Customer, higher on Orders). The local views now satisfy the consistency requirement and orders_prj also satisfies the currency bound but cust_prj will never be current enough to be useful. Thus a mixed plan (plan 4) was selected by the optimizer. If we relax the currency bound on Customer further as in Q5, both local views becomes usable and plan 5 is selected. Q3, Q4 and Q5 demonstrate how changing the currency can drastically change the query plan.

As we can see in Figure 4.9, every local data access is protected by a currency guard, which guarantees that local data that is too stale will never be used.

Optimization is entirely cost based. One consequence of this is that the optimizer may choose not to use a local view even though it satisfies all requirements if it is cheaper to get the data from the back-end server. This is illustrated by the following two queries. Even though they differ only in their range predicates, the optimizer chooses different plans for them.

For Q6, a remote query was chosen even though the local view cust_prj satisfied the currency requirement. The reason is the lack of a suitable secondary index on cust_prj while there is one at the back-end server. The range predicate in Q6 is highly selective (53 rows returned) so the index on c_acctbal at the back-end is very effective, while at the cache the whole view (150,000 rows) would have to be scanned. When we increase the range, as in Q7, the benefit of an index scan over a sequential scan diminishes and a plan exploiting the local view is chosen.

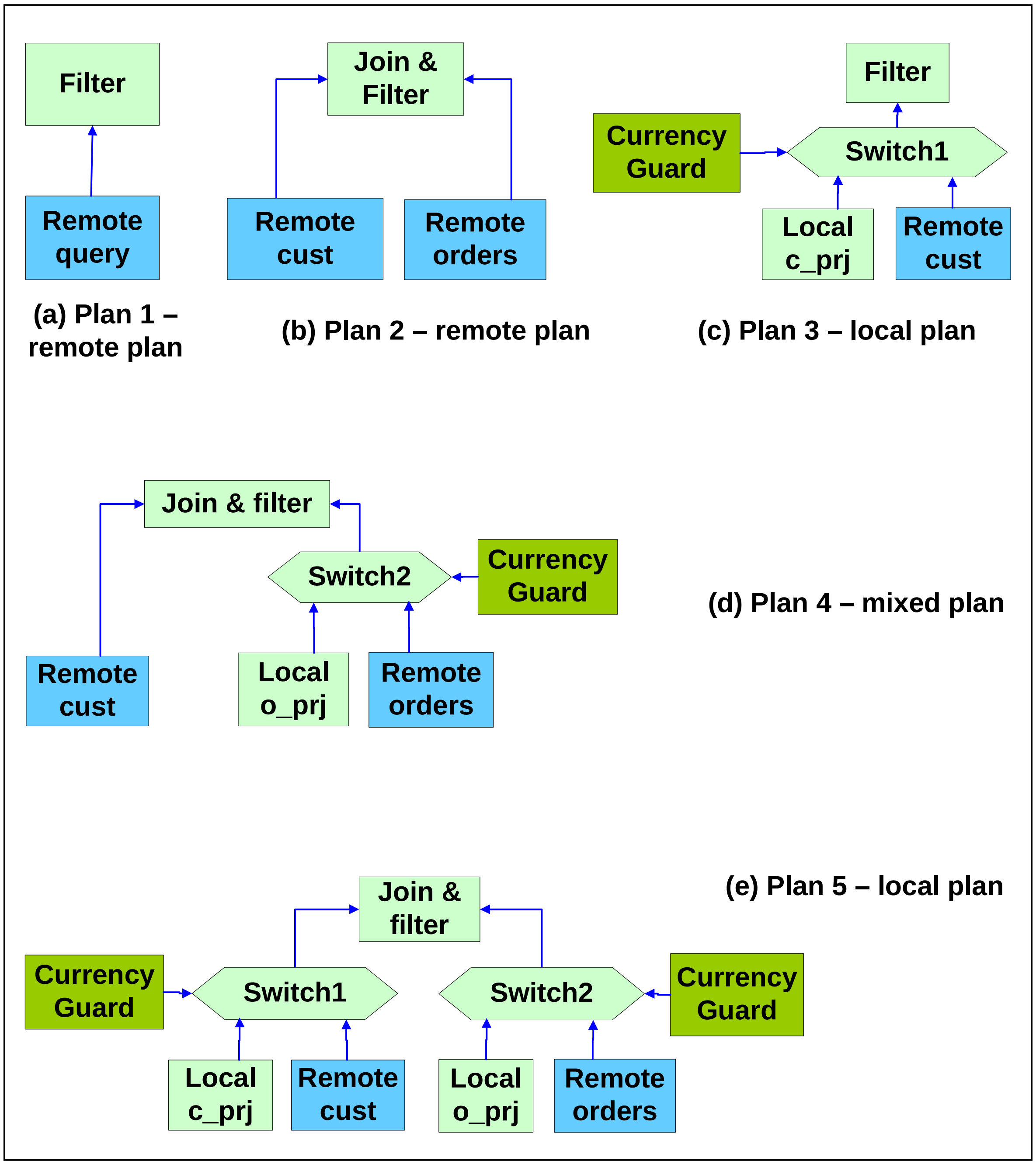


Figure 4.9 Generated logical plans

4.4.4 Currency Guard Overhead

To guarantee that the result satisfies the query's currency bounds, the optimizer generates plans with a currency guard for every local view in the plan. What is the actual overhead of currency guards in the current system implementation? Where does time go? We ran a series of experiments aimed at answering these questions using the queries shown in Table 4.2.

Qa is the simplest and fastest type of query but also very common in practice. The local cache and the back-end server used the same trivial plan: lookup on the clustering index. For Qb, both servers used the same plan: a nested loop join with orders_prj (Orders) as the (indexed) inner. Again, for Qc, both servers used the same plan: a complete table scan.

Qa:
key select
	SELECT c_custkey, c_name, c_nationkey,
	c_acctbal, c_mktsegment
	FROM customer
	WHERE c_custkey = 1
	[CURRENCY 10 on (customer)]

Qb:
join query
	SELECT c_custkey, c_name, o_orderkey,
	o_totalprice
	FROM customer, orders
	WHERE c_custkey=o_custkey and c_custkey=1
	[CURRENCY 10 on (customer), 20 on (orders)]

Qc:
non-key
select
	SELECT c_custkey, c_name, c_nationkey
	c_acctbal, c_mktsegment
	FROM customer
	WHERE c_nationkey = 1
	[CURRENCY 10 on (customer)]

Table 4.2 Queries used for experiments

	Local			Remote		
	Qa	Qb	Qc	Qa	Qb	Qc
Cost (ms)	0.11	0.19	2.39	0.24	0.42	0.90
Cost (%)	15.25	21.30	3.66	3.59	4.31	0.41
Base query (ms)	0.72	0.89	65.30	6.69	9.74	219.51

Table 4.3 Currency guard overhead

For each query, we generated two traditional plans without currency checking (one local and one remote) and a plan with currency checking. We ran the plan with currency checking twice, once with the local branches being executed and the other with the remote branches being executed. We then compared their execution times (elapsed time) with the execution times of the plans without currency guards. In each run, we first warmed up the cache, then executed the current query repeatedly (100,000 times for Qa and Qb local execution, 1000 for Qc remote execution and 1000 for the others) and computed the average execution time. Note that we executed exactly the same query in order to reduce buffer pool and cache misses, thereby minimizing the execution time (and maximizing the relative overhead). The last row of Table 4.3 shows the execution time for the queries without run-time currency checking. The rest of the table shows the absolute and relative cost of currency guards and the number of output rows.

In absolute terms, the overhead is small, being less than a millisecond for Qa and Qb. In the remote cases the relative overhead is less than 5% simply due to longer execution times. However, in the local case the relative overhead of 15% for Qa and 21% for Qb seems surprisingly high, even taking into account their very short execution time.

	setup		run		shutdown		IdealTotal	
	ms	%	ms	%	ms	%	ms	%
Qa	0.04	27.13	0.06	152.52	0.01	26.56	~0.07	~11.51
Qb	0.06	39.39	0.09	98.52	0.01	29.69	~0.10	~14.32
Qc	0.01	2.98	1.99	3.79	0.04	46.21	~0.10	~0.16

Table 4.4 Local overhead breakdown

Where does the extra time go? We investigated further by profiling the execution of local plans. The results are shown in the first three columns of Table 4.4 with each column showing an absolute overhead and a relative overhead. Each column corresponds to one of the main phases during execution of an already-optimized query: setup plan, run plan and shutdown plan. The absolute difference for a phase is the difference between the (estimated) elapsed time for the phase in the plan with and without currency checking. The relative difference is as a percentage of the time of that phase in the plan without currency checking. In other words, both indicate how much the elapsed time of a phase has increased in the plans with currency checking.

During the setup phase, an executable tree is instantiated from the query plan, which also involves schema checking and resource binding. Compared with a traditional plan, a plan with currency checking is more expensive to set up because the tree has more operators and remote binding is more expensive than local binding. From Table 4.4, we see that the setup cost of a currency guard is independent of the output size but increases with the number of currency guards in the plan. For small queries such as Qa and Qb, the overhead for this phase seems high. We found that the overhead is not inherent but primarily caused by earlier implementation choices that slow down setup for SwitcshUnions with currency guards. The problem has been diagnosed but not yet remedied.

During the run phase, the actual work of processing rows to produce the result is done. The overhead for Qa and Qb is relatively high because running the local plans is so cheap (Single indexed row retrieval for Qa, and 6-row indexed nested loop join for Qb). The overhead for a SwitchUnion operator during this phase consists of two parts: evaluating the guard predicate once and overhead for each row passing through the operator. Evaluating the predicate is done only once and involves retrieving a row from the local heartbeat table and applying a filter to it. Qa just retrieves a single row from the Customer table so it is not surprising that the relative overhead is as high as it is. In Qc, almost 6000 rows pass through the SwitchUnion operator so the absolute overhead increases but the relative overhead is small, under 4%. There are some (limited) opportunities for speeding up this phase.

In an ideal scenario (i.e., with possible optimizations in place), it should be possible to reduce the overhead of a currency guard to the overhead in Qa plus the shutdown cost. Based on this reasoning, we estimated the minimal overhead for our workload. The results are shown in the IdealLocal column of Table 4.4.

Chapter 5

Enforcing Data Quality Constraints for Finer Granularity

The extension to finer granularity cache management fundamentally changes every aspect of the problem, imposing non-trivial challenges: 1) how the cache tracks data quality; 2) how administrators specify cache properties; 3) how to maintain the cache efficiently; and 4) how to do query processing. While Chapter 3 addresses the first three questions, in this chapter we focus on the last one. A traditional distributed query optimizer decides whether to use local data based on data availability and estimated cost. In our setting, it must also take into account local data properties (presence, consistency, completeness and currency). Presence checking is addressed in [ZLG05]; the same approach can be extended to completeness checking. This chapter describes efficient checking for C&C constraints, the approach being an extension of the framework developed in Chapter 4. Theorem 5.1, Theorem 5.2, and Theorem 5.3 guarantee the correctness of our algorithms.

The algorithms developed here are generalizations of those in Chapter 4 to cover finer granularity C&C checking. In Chapter 4, consistency checking was done completely at optimization time and currency checking at run time, because view level cache region information is stable and available at optimization, while currency information is only available at run time. In this chapter we still perform as much consistency checking as possible at optimization time but part of it may have to be delayed to run time. For a view with partial consistency guarantees, we don't know at optimization time which actual groups will be consistent at run time. Further, ad-hoc cache regions may change over time, also prompting run-time checking.

The rest of the chapter is organized as follows. Sections 5.1 to 5.3 illustrate the extensions to C&C constraints normalization, query compilation time consistency checking, and query execution time C&C checking respectively. Experimental and analytical results are reported in Section 5.4.

5.1 Normalizing C&C Constraints

A query may contain multiple currency clauses, at most one per SFW block. The first task is to combine the individual clauses and convert the result to a normal form. To begin the process, each currency clause is represented as follows.

Definition: (**Currency and consistency constraint**) A C&C constraint CCr is a set of tuples, CCr = {<b_1, $\mathbf{K}_1$, $\mathbf{S}_1$, $\mathbf{G}_1$>, ..., <b_n, $\mathbf{K}_n$, $\mathbf{S}_n$, $\mathbf{G}_n$>}, where $\mathbf{S}_i$ is a set of input operands (table or view instances), b_i is a currency bound specifying the maximal acceptable staleness of the input operands in $\mathbf{S}_i$, $\mathbf{G}_i$ is a grouping key and $\mathbf{K}_i$ a set of grouping key values. □

Each tuple has the following meaning: for any database instance, if we group the input operands referenced in a tuple by the tuple's grouping key $\mathbf{G}_i$, then for those groups with one of the key values in $\mathbf{K}_i$, each group is consistent. The key value sets $\mathbf{K}_i$ will be used when constructing consistency guard predicates to be checked at run time. Note that the default value for each field is the strongest constraint.

All constraints from individual currency clauses are merged together into a single constraint and converted into an equivalent or stricter normalized form with no redundant requirements.

Definition: (**Normalized C&C constraint**) A C&C constraint CCr = {$<b_1, \mathbf{K}_1, \mathbf{S}_1, \mathbf{G}_1>$, ..., $<b_n, \mathbf{K}_n, \mathbf{S}_n, \mathbf{G}_n>$} is in normalized form if all input operands (in the sets $\mathbf{S_i}$) are base tables and the input operand sets $\mathbf{S}_1, \ldots, \mathbf{S}_n$ are all non-overlapping. □

We extend the algorithm for transforming a set of constraints into normalized form given in Figure 4.1 to cover finer granularity. The structure of the algorithm remains unchanged, while the rules are generalized. As shown in Figure 4.1, the algorithm first recursively expands all references to views into references to base tables. Next, it repeatedly merges any two tuples that have one or more input operands in common, but using the following generalized rule. Refer to [RG02] for the concept of *attribute closure*.

Normalization Rule: Given CCr_1 = {$<b_1, \mathbf{K}_1, \mathbf{S}_1, \mathbf{G}_1>$} and CCr_2 = {$<b_2, \mathbf{K}_2, \mathbf{S}_2, \mathbf{G}_2>$}, $\mathbf{S}_1 \cap \mathbf{S}_2 \neq \emptyset$, replace the two constraints by CCr = {$<b, \mathbf{K}, \mathbf{S}, \mathbf{G}>$}, where $b = \min(b_1, b_2)$, and $\mathbf{S} = \mathbf{S}_1 \cup \mathbf{S}_2$. Given a set of functional dependencies (FDs) F over the query result relation Y, let $\mathbf{G}_i^+$ be the attribute closure of $\mathbf{G_i}$ w.r.t. F, where i = 1, 2. Then $\mathbf{G} = \mathbf{G}_1^+ \cap \mathbf{G}_2^+$. Let $\mathbf{K_i}^+ = \Pi_G \sigma_{Gi=ki}(Y)$, i = 1, 2. Then $\mathbf{K} = \mathbf{K_1}^+ \cup \mathbf{K_2}^+$. □

Given a set of FDs over the base relations, and the equivalence classes induced by a query, we can infer the set of FDs over the query result relation. For example, for Q2, let CCr_1 = {<10, Ø, {Authors, Books}, {city}>}, CCr_2 = {<5, Ø, {Books}, {isbn}>}. CCr_1 requires that if we group the query result by city, then within each group, all the rows have to be consistent. CCr_2 requires that if we group the result by isbn, then each book row has to be consistent. From the key constraints in Authors and Books, together with the join condition in Q2, we know that isbn is a key for the final relation. Thus CCr = {<5, Ø, {Authors, Books}, {city}>}. If an instance satisfies CCr, then it must satisfy both CCr_1 and CCr_2, and vice versa.

In what follows, we formally define implication and equivalence between any two CCrs, and prove that when K1 and K2 are set to default, then the outcome of the normalization rule CCr is equivalent to the inputs $CCr_1 \cup CCr_2$ w.r.t. F. Further, we prove that not knowing all FDs doesn't affect the correctness of the rule.

Definition: (**Implication, Equivalence**) Given two C&C constraints CCr_1 and CCr_2, a cache schema Λ, and a set of FDs F over Λ**,** we say that CCr_1 implies CCr_2 w.r.t Λ and F, if every instance of Λ that satisfies F and CCr_1 also satisfies CCr_2. If CCr_1 implies CCr_2 w.r.t Λ and F and CCr_2 implies CCr_1 w.r.t Λ and F, then CCr_1 and CCr_2 are equivalent w.r.t Λ and F. □

Lemma 5.1: Given a cache schema Λ, for any CCr = {<b, **K**, **S**, **G**>} and any instance of Λ, the consistency constraint in CCr can be satisfied w.r.t. Λ and F, iff the grouping key **G'** of the cache region partitioning on **S** in Λ is a subset of $\mathbf{G}^+$ w.r.t. Λ and F. □

Proof of Lemma 5.1:

Sufficiency is obvious. Now we prove necessity. Since each group by grouping key **G** belongs to one group by grouping key **G’**, **G** functionally determines **G’**. Thus **G’** $\subseteq$ **G**$^+$.

Theorem 5.1: If $\mathbf{K}_1$ and $\mathbf{K}_2$ are set to default, then the output of the Normalization Rule CCr is equivalent to its input $CCr_1 \cup CCr_2$ w.r.t. Λ and F. □

Proof of Theorem 5.1:

Given any instance of Λ that satisfies {CCr} w.r.t. to F, from Lemma 5.1, the grouping key of its cache region partitioning is a subset of $\mathbf{G}^+$. Since $\mathbf{G} \subseteq \mathbf{G_i}^+$, i = 1, 2, $\mathbf{G}^+ \subseteq \mathbf{G_i}^+$**,** the consistency constraints in ($CCr \cup CCr_2$} are satisfied. Further, since the consistency partitioning satisfies currency constraint b, and b = min (b_1, b_2), b_1 and b_2 are also satisfied. □

From Lemma 5.1, it follows that for any instance that satisfies both CCr_1 and CCr_2 w.r.t. F, the grouping key of its cache region partitioning has to be a subset of **G**. Thus, it also satisfies CCr. Since it satisfies b_1 and b_2, and b = min(b_1, b_2), it also satisfies b.

Theorem 5.2: Suppose FDs over a cache schema Λ: $F+ \subset F'+$. The output of the Normalization Rule {CCr} w.r.t. F implies its input $CCr_1 \cup CCr_2$ w.r.t. Λ and F’. □

Proof of Theorem 5.2:

Let $\mathbf{G} = \mathbf{G}_1^+ \cap \mathbf{G}_2^+$ w.r.t. F, $\mathbf{G'} = \mathbf{G}_1^+ \cap \mathbf{G}_2^+$ w.r.t. F'. Then $\mathbf{G} \subseteq \mathbf{G'}$. Thus for any instance of Λ that satisfies CCr, since $\mathbf{K} = \mathbf{K_1}^+ \cup \mathbf{K_2}^+$ w.r.t. F, from Lemma 5.1, it satisfies $CCr_1 \cup CCr_2$. □

5.2 Compile-time Consistency Checking

We take the following approach to consistency checking. At optimization time, we proceed as if all consistency guarantees were full. A plan is rejected if it would not produce a result satisfying the query's consistency requirements even under that assumption. Whenever a view with partial consistency guarantees is included in a plan, we add consistency guards that check at run-time if the guarantee holds for the groups actually used.

As explained in Chapter 4 SQL Server uses a transformation-based optimizer. Conceptually, optimization proceeds in two phases: an exploration phase and an optimization phase. The former generates new logical expressions; the latter recursively finds the best physical plan. Physical plans are built bottom-up.

Required and delivered (physical) plan properties play a very important role during optimization. To make use of the plan property mechanism for consistency checking, we must be able to perform the following three tasks: 1) transform the query's consistency constraints into required consistency properties; 2) given a physical plan, derive its delivered consistency properties from the properties of the local views it refers to; 3) check whether delivered consistency properties satisfy required consistency properties.

5.2.1 Required Consistency Plan Property

A query's required consistency property consists of the normalized consistency constraint described in Section 5.1.

5.2.2 Delivered Consistency Plan Property

A delivered consistency property CPd consists of a set of tuples {<R_i, $\mathbf{S}_i$, $\mathbf{\Omega}_i$>} where R_i is the id of a cache region, $\mathbf{S}_i$ is a set of input operands, namely, the input operands of the current expression that belong to region R_i, and $\mathbf{\Omega}_i$ is the set of grouping keys for the input operands. Each operator computes its delivered plan properties bottom-up based on the delivered plan properties of its inputs. We extend the algorithms shown in Figure 4.2 and Figure 4.3 for calculating derived C&C property to cover the generalized form of derived C&C property. The structure of the algorithms remains unchanged, while the rules are generalized, as stated shortly. For convenience, we call the extended one *Algorithm DrvdPropertyCalcGeneral*.

Delivered plan properties are computed bottom-up for each physical operator, in terms of the properties of its inputs, according to the algorithm described in Figure 4.2, which treats the physical operators accordingly as four categories: i) leaves of the plan tree (e.g., tables or materialized views), ii) single-input operators, iii) joins, and iv) SwitchUnion.

The *leaves of a plan tree* are table, materialized view, or index scan operators, possibly with a range predicate. If the input operand is a local view, return the ids of the view's input tables in **S**, not the id of the view, since consistency properties always refer to base tables. If the whole view is consistent, the id of its cache region; otherwise, return the set of grouping keys of its consistency root, and a flag, say –1, in the region id field to indicate row-level granularity. For a remote table or view, return a special region id, say, 0.

All *operators with a single relational input*, such as filter, project, aggregate and sort do not affect the delivered consistency property so copy the property from their relational input.

Join operators combine multiple input streams into a single output stream. Union the input consistency properties and merge property tuples that are in the same cache region. Formally, given two delivered C&C property tuples $CPd_1 = \{<R_1, \mathbf{S}_1, \mathbf{\Omega}_1>\}$ and $CPd_2 = \{<R_2, \mathbf{S}_2, \mathbf{\Omega}_2>\}$, merge them if either of the following conditions is true:

1) If the input operands are from the same cache region, i.e., $R_1 = R_2 \geq 0$, then merge the tables, i.e., replace CPd_1 and CPd_2 by $CPd = \{<R_1, \mathbf{S}, Ø >\}$, where $\mathbf{S} = \mathbf{S}_1 \cup \mathbf{S}_2$.
2) If the input operands are grouped into cache regions by the same keys (for the same root), i.e., $\mathbf{\Omega}_1 = \mathbf{\Omega}_2$, they are group-wise consistent so merge them into $CPd = \{<-1, \mathbf{S}, \mathbf{\Omega}_1>\}$ where $\mathbf{S} = \mathbf{S}_1 \cup \mathbf{S}_2$. □

A *SwitchUnion* operator has multiple input streams but it does not combine them in any way; it simply selects one of the streams. Thus, the output consistency property is the strongest consistency property implied by every input. When generating an intersection from two derived C&C phrases as shown in Figure 4.3, we follow the generalized rules. Formally, given two delivered C&C property tuples $<R_1, \mathbf{S}_1, \mathbf{\Omega}_1>$ and $<R_2, \mathbf{S}_2, \mathbf{\Omega}_2>$, where $\mathbf{S}_1 \cap \mathbf{S}_2 \neq \emptyset$, generate a common C&C tuple $<R, \mathbf{S}, \mathbf{\Omega}>$ if any of the following conditions is true:

1) If $\mathbf{\Omega}_1 = \mathbf{\Omega}_2 = \emptyset$, then $\mathbf{\Omega} = \emptyset$, and R is determined the same way as in Figure 4.3.

2) If one and only one of $\mathbf{\Omega}_i = \emptyset$, i = 1, 2, without loss of generality, suppose $\mathbf{\Omega}_1 = \emptyset$, then $R = R_2$, $\mathbf{\Omega} = \mathbf{\Omega}_2$.
3) If $\mathbf{\Omega}_1^+ \cap \mathbf{\Omega}_2^+ \neq \emptyset$, where $\mathbf{\Omega}_i^+$ denotes the union of the attribute closure of each grouping key in $\mathbf{\Omega}_i$, i=1, 2, then $R = -1$, and $\mathbf{\Omega} = \mathbf{\Omega}_1^+ \cap \mathbf{\Omega}_2^+$.

For all cases $\mathbf{S} = \mathbf{S}_1 \cup \mathbf{S}_2$. □

In case 1), both tuples are view level consistent, thus the common tuple is generated following the same rule as in Figure 4.3. In case 2), only one tuple is view level consistent, since group level consistency is weaker, we output the group level consistency guarantee. In case 3), both tuples are group level consistent, thus we can only output the common grouping keys implied by them.

5.2.3 Satisfaction Rules

Now, given a required consistency property CCr and a delivered one CPd, how do we know whether CPd satisfies CCr? Firstly, our consistency model does not allow two columns from the same input table T to originate from different snapshots, leading to the following property:

Conflicting consistency property: A delivered consistency property CPd is conflicting if there exist two tuples $<R_1, \mathbf{S}_1, \mathbf{\Omega}_1>$ and $<R_2, \mathbf{S}_2, \mathbf{\Omega}_2>$ in CPd s.t. $\mathbf{S}_1 \cap \mathbf{S}_2 \neq \emptyset$ and one of the following conditions holds: 1) $R_1 \neq R_2$, or 2) $\mathbf{\Omega}_1 \neq \mathbf{\Omega}_2$. □

This property is conservative in that it assumes that two cache regions U_1 and U_2 from different views can only be consistent if they have the same set of control-keys.

Secondly, a complete plan satisfies the constraint if each required consistency group is fully contained in some delivered cache region. We extend the consistency satisfaction rule in Chapter 4 to include finer granularity cache regions.

Consistency satisfaction rule: A delivered consistency property CPd satisfies a required CCr w.r.t. a cache schema Σ and functional dependencies F, iff CPd is not conflicting and, for each tuple $<b_r, \mathbf{K}_r, \mathbf{S}_r, \mathbf{G}_r>$ in CCr, there is a tuple $<R_d, \mathbf{S}_d, \mathbf{\Omega}_d>$ in CPd s.t. $\mathbf{S}_r \subseteq \mathbf{S}_d$, and one of the following conditions holds: 1) $\mathbf{\Omega}_d = \emptyset$, or 2) there exists a $\mathbf{G}_d \in \mathbf{\Omega}_d$ s.t. $\mathbf{G}_d \subseteq \mathbf{G}_r^+$ where $\mathbf{G}_r^+$ is the attribute closure of $\mathbf{G}_r$ w.r.t. F. □

For query Q2, suppose we have CCr = {<5, Ø, {Authors, Books}, {isbn}>}, and that the cache schema is the one in Figure 3.6. During view matching, AuthorCopy and BookCopy will match Q2. Thus CPd = {<–1, {Authors, Books}, {Authors.authorId, city}>}. If AuthorCopy joins with BookCopy on authorId (a join condition indicated by the presence correlation), and the result is R, then from the key constraints of Authors and Books we know that isbn is a key in R. Therefore city$\in$ {isbn}$^+$. CPd satisfies CCr.

Not knowing all FDs doesn’t affect the correctness of the satisfaction rule, it only potentially produces false negatives:

Theorem 5.3: For any two sets of functional dependencies F and F' over the cache schema Σ, where $F^+ \subseteq F'^+$, if a delivered consistency property CPd satisfies a required CCr w.r.t. F, then it satisfies CCr w.r.t. F'. □

Proof of Theorem 5.3:

Let $\mathbf{G}_r^+$ be the attribute closure of $\mathbf{G}_r$ w.r.t. F^+, $\mathbf{G}_r'^+$ be the attribute closure of $\mathbf{G}_r$ w.r.t. F'^+, then $\mathbf{G}_r^+ \subseteq \mathbf{G}_r'^+$. □

Theorem 5.4: Assuming run-time checking is correct, with the Delivered-Plan Algorithm, for any plan of which CPd satisfies CCr w.r.t. a cache schema Σ and functional dependencies F, no matter which data sources are used at execution time, CCr will be satisfied w.r.t F. □

Proof of Theorem 5.4:

Let the set of C&C properties of the sources be CPd = $\{< R_{di}, \mathbf{S}_{di}, \mathbf{\Omega}_{di} >\}$. Let the output of the Delivered-Plan Algorithm be CPd'.

Case 1: There are no SwitchUnion operators in the plan.

Since operators with a single relational input simply pass the input property; while join operators simply merge the input properties with the same cache region, we have CPd = CPd'.

Case 2: There are some SwitchUnions used as C&C guards.

In this case, for each SWU, there are two types of checking: consistency checking and currency checking. So the branch actually used satisfies both consistency and currency constraints.

The difference between CPd and CPd' is that in CPd, for a local source with property $CPd_i = \{< R_{di}, \mathbf{S}_{di}, \mathbf{\Omega}_{di}>\}$ guarded with a SWU, we have either CPd_i or $CPd_i' = \{<0, \mathbf{S}_{di}, \emptyset>\}$, depending on whether the local branch or the remote branch is used during execution.

For any tuple $r = <b_r, \mathbf{K}_r, \mathbf{S}_r, \mathbf{G}_r>$ in CCr, since CPd' satisfies CCr, there exists a row $t = <R_d, \mathbf{S_d}, \mathbf{\Omega_d}>$, such that, $\mathbf{S}_r \subseteq \mathbf{S}_d$, and one of the following conditions holds: i) $\mathbf{\Omega}_d = \emptyset$, or ii) let $\mathbf{G}_r$+ be the attribute closure w.r.t. F. There exists a $\mathbf{G}_d \in \mathbf{\Omega}_d$ such that $\mathbf{G}_d \subseteq \mathbf{G}_r^+$.

If t is merged from sources that don't have a SWU, then it also appears in CPd, otherwise, w/o loss of generality, we can assume it comes from two local resources with SWU operators and with property $t_1 = < R_{d1}, \mathbf{S}_{d1}, \mathbf{\Omega}_{d1}>$ and $t_2 = < R_{d2}, \mathbf{S}_{d2}, \mathbf{\Omega}_{d2}>$.

Trivial case: If $\mathbf{S}_r \subseteq \mathbf{S}_{d1}$(or $\mathbf{S}_{d2}$), then r is satisfied by t_1 (or t_2) in CPd.

Otherwise, we claim that for any cache instance, either both local branches are used or both remote branches are used. Thus if CPd' satisfies CCr, then if we plug in CPd the property of the data sources actually used, CPd also satisfies CCr.

Case 2.1: R>0. Since both local resources belong to the same cache region, they have the same currency, so does the currency checking result.

Case 2.2: R= –1. Since the two resources are controlled by the same set of consistency control-keys, again, the C&C checking results are the same. □

While a plan is being constructed, bottom-up, we want to stop as soon as the current subplan cannot deliver the consistency required by the query. The consistency satisfaction rule cannot be used for checking subplans; a check may fail simply because the partial plan does not include all inputs covered by the required consistency property. Instead we apply the following *violation rules*. We prove that a plan cannot satisfy the required plan properties if a subplan violates any of the three rules (Theorem 5.5).

Consistency violation rules: A delivered consistency property CPd violates a required consistency constraint CCr w.r.t. a cache schema Σ and functional dependencies F, if one of the following conditions holds:

1) CPd is conflicting,
2) There exists a tuple $< b_r, \mathbf{K}_r, \mathbf{S}_r, \mathbf{G}_r >$ in CCr that intersects more than one consistency group in CPd, that is, there exist two tuples $< R1_d, \mathbf{S1}_d, \mathbf{\Omega 1}_d >$ and $< R2_d, \mathbf{S2}_d, \mathbf{\Omega 2}_d >$ in CPd s.t. $\mathbf{S}_r \cap \mathbf{S1}_d \neq \emptyset$ and $\mathbf{S}_r \cap \mathbf{S2}_d \neq \emptyset$,
3) There exists $<b, \mathbf{K}_r, \mathbf{S}_r, \mathbf{G}_r>$ in CCr, and $< R_d, \mathbf{S}_d, \mathbf{\Omega}_d >$ in CPd, s.t. $\mathbf{S}_r \subseteq \mathbf{S}_d$, $\mathbf{\Omega}_d \neq \emptyset$ and the following condition holds: let $\mathbf{G}_r+$ be the attribute closure w.r.t. Σ and F. There does <u>not</u> exist $\mathbf{G}_d \in \mathbf{\Omega}_d$, s.t. $\mathbf{G}_d \subseteq \mathbf{G}_r^+$. □

Theorem 5.5: Using the Delivered-Plan Algorithm, if a partial plan A violates the required consistency property CCr w.r.t. a cache schema Σ and functional dependencies F, then no plan that includes A as a branch can satisfy CCr w.r.t. Σ and F . □

Proof of Theorem 5.5:

This is true because from the algorithm, for any tuple < R_d, $\mathbf{S}_d$, $\mathbf{\Omega}_d$ > in the delivered plan property of P, there is a tuple < R_d, $\mathbf{S}_d$', $\mathbf{\Omega}_d$ > in the delivered plan property of any plan that includes P as a branch, where $\mathbf{S}_d \subseteq \mathbf{S}_d$'. □

5.3 Run-time Currency and Consistency Checking

To include C&C checking at run-time, the optimizer must produce plans that check whether a local view satisfies the required C&C constraints and switch between using the local view and retrieving the data from the backend server. Such run-time decision-making is built into a plan by using a *SwitchUnion* operator. A SwitchUnion operator has multiple input streams but only one is selected at run-time based on the result of a selector expression.

In MTCache, all local data is defined as materialized views and logical plans making use of a local view are always created through view matching [LGZ04, GL01]. Consider an (logical) expression E and a matching view V from which E can be computed. If C&C checking is required, we produce a substitute consisting of a SwitchUnion on top, as shown in Figure 5.1, with a selector expression that checks whether V satisfies the currency and consistency constraint, and two input expressions: a local branch and a remote branch. The

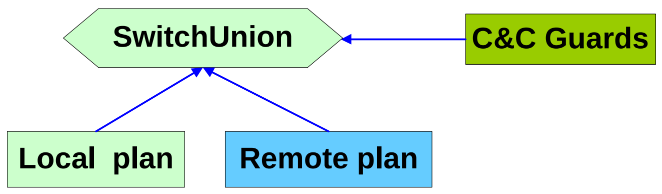


Figure 5.1 SwitchUnion with consistency and currency guards

local branch is a normal substitute expression produced by view matching and the remote plan consists of a remote SQL query created from the original expression E. If the condition, which we call consistency guard or currency guard according to its purpose, evaluates to true, the local branch is chosen, otherwise the remote one.

The discussion of when and what type of consistency checking to generate and the inexpensive consistency checking we support is deferred to Section 5.4.

Currency bound checking: If the required lowest currency bound on the input tables of E is B, the optimizer generates a currency guard that checks if any required region is too stale for the query. Given a control-table CT on control-key CK, a set of probing values **K** on CK, the check is:

```
NOT EXIST (     SELECT 1 FROM CT
                WHERE CK IN K AND rid < getdate() – B   )
```

Recall that the timestamp is recorded in the rid column of each control-table (Section 3.4.1.2).

5.4 Analysis and Experiments

This section reports experimental results for consistency checking; results for presence and currency checking are reported in [ZLG05] and Chapter 4 respectively.

5.4.1 Experimental Setup

We used a single cache DBMS and a backend server. The backend server hosted a TPCD database with scale factor 1.0 (about 1GB), where only the Customers and Orders tables were used. The Customers table was clustered on its primary key, c_custkey with an index on c_nationkey. The Orders table was clustered on (o_custkey, o_orderkey). The cache had a copy of each table, CustCopy and OrderCopy, with the same indexes. The control table settings and queries used are shown in Figure 5.2. We populated the ckey and nkey columns with c_custkey and c_nationkey columns from the views respectively.

C_PCT and O_PCT are the presence control tables of CustCopy and OrderCopy

```
Settings: CREATE TABLE C_PCT (ckey int PRIMARY, rid int)
          CREATE TABLE C_CsCT(nkey int PRIMARY, rid int)
          CREATE TABLE O_PCT (ckey int PRIMARY, rid int)

Qa:     SELECT  *
        FROM    Customers C
        WHERE   c_custkey in $custSet
        [CURRENCY BOUND 10 on (C) BY $key]

Qb:     SELECT  *
        FROM    Customers C, orders O
        WHERE   c_custkey=o_custkey and c_custkey IN $custSet
        [CURRENCY BOUND 10 on (C, O) BY $key]

Qc:     SELECT  *
        FROM    Customers C
        WHERE   c_nationkey in $nationSet
        [CURRENCY 10 on (C) BY $key]
```

Figure 5.2 Settings and queries used for experiments

respectively. C_CsCT is a consistency control table on CustCopy. By setting the timestamp field, we can control the outcome of the consistency guard.

The caching DBMS ran on an Intel Pentium 4CPU 2.4 GHz box with 500 MB RAM. The backend ran on an AMD Athlon MP Processor 1800+ box with 2GB RAM. Both machines ran Windows 2000 and were connected by LAN.

5.4.2 Success Rate of Ad-hoc Consistency Checking (Analytical)

Intuitively, everything else being equal, the more relaxed the currency requirements are, the more queries can be computed locally. Although less obvious, this is also true for consistency constraints.

Assuming all rows in $custSet are in the cache, a dynamic plan for Qa will switch between either CustCopy and a remote query, depending on the outcome of the consistency guard. If there is only one customer in $custSet, by default the result is consistent. At the other extreme, if $custSet contains 1000 customers, they are not likely to be consistent. When the number of customers in $custSet increases, the likelihood of the result being consistent decreases. Suppose there are N rows in CustCopy, divided into M cache regions. We assume that the regions are the same size and each row in $custSet is independently and randomly chosen from CustCopy. Let the size of $custSet be x, where $x \leq N$. The result is consistent only when all the chosen rows are from the same cache region. Thus, the probability of an ad-hoc consistency check being successful is $P(consistent) = (1/M)^{X-1}$.

As one would expect, and as this formula confirms, the probability of success decreases rapidly as the number of consistency groups and/or number of required rows increase (Figure 5.3 (a)).

5.4.3 Update Cost (Analytical)

Suppose there are N rows in CustCopy, divided into M cache regions. We assume that the regions are the same size and each row in $custSet is independently and randomly chosen from CustCopy. We model the cost of refreshing a consistency region in CustCopy once by three components: C_{setup}, setup cost of the remote query; C_{calc}, calculation cost at the backend; $C_{transfer}$, network transfer cost for the query results.

$$C_{refresh\text{-}once} = C_{setup} + C_{calc} + C_{transfer}$$

In this formula, C_{setup} is a fixed setup cost for every query, while the other two are proportional to the number of tuples. Thus, $C_{calc} + C_{transfer}$ can be expressed as $C_{calc} + C_{transfer} = \frac{N}{M} * C_{unit}$, where C_{unit} is the cost of calculating and transferring one tuple. Let the refresh frequency be f, then in a time unit, the total refresh cost for the region will be:

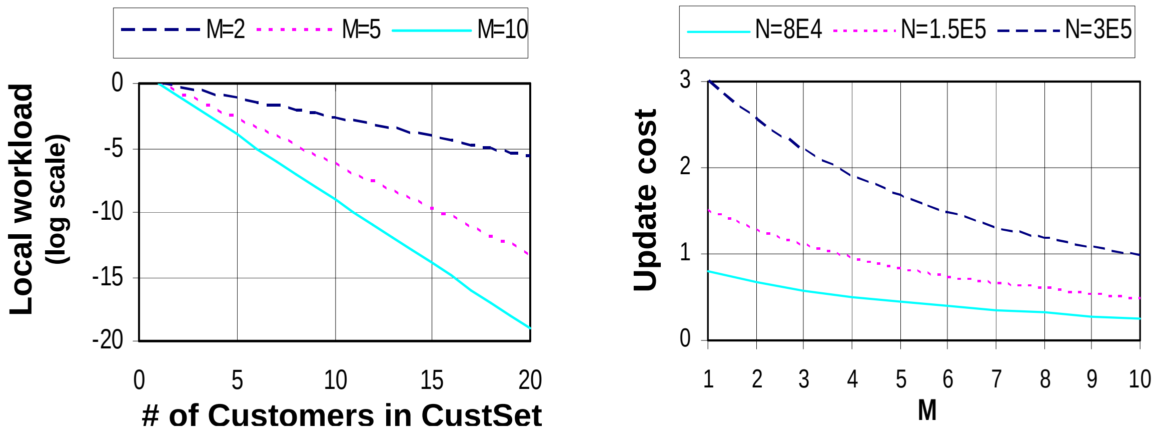
$$C_{refresh} = f * C_{refresh\text{-}once}$$

By dividing data into different consistency regions, we have the advantage of being able to refresh them at different rates. For example, update platinum customers every 10 minutes, and normal customers every hour. To model this, we use a decrease factor r. Suppose that originally all the rows are in one region, with refresh rate 1. For each consistency region added, the refresh rate drops by r% from its predecessor, i.e., for region i, $f_i = (1 - r)^{i-1}$. Thus, the total update cost of refreshing M regions is as follows, and the function is plotted in Figure 5.3 (b).

$$C_{refresh-total} = \sum_{i=1}^{M} f_i * (C_{setup} + \frac{N}{M} * C_{unit})$$
$$= (C_{setup} + \frac{N}{M} * C_{unit}) \sum_{i=1}^{M} f_i$$

5.4.4 Consistency Guard Overhead

We made the design choice to only support certain inexpensive types of run-time consistency guards. A natural question is: what is the overhead of the consistency guards? Furthermore, how expensive are more complicated guards?



(a) Probability of successful ad-hoc consistency check

(b) Update cost of refreshing M regions

Figure 5.3 Workload shift and update cost

We experimentally evaluate the cost of a spectrum of guards by means of emulation. Given a query Q, we generate another query Q' that includes a consistency guard for Q, and use the execution time difference between Q' and Q to approximate the overhead of the consistency guard. For each query, depending on the result of the consistency guard, it can be executed either locally or at the backend. We measure the overhead for both scenarios.

5.4.4.1 Single-Table Case

We first analyze what type of consistency guard is needed for Qa when $key differs. The decision making process is shown in Figure 5.4 and the consistency guards in Figure 5.5.

Condition A: Is each required consistency group equal to or contained in a presence region?

If Yes, it follows from the Presence Assumption that all the rows associated with each presence control-key are consistent. No explicit consistency guard is needed. For example, for Qa with $key = c_custkey.

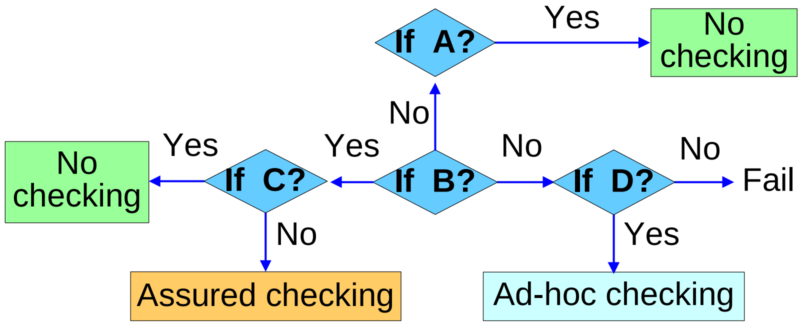


Figure 5.4 Generating consistency guard

Condition B: Is each required consistency group equal to or contained by a consistency region? If Yes, we check C, otherwise we check D.

Condition C: Is the consistency guarantee full?

If Yes, then no run-time consistency checking is necessary. Otherwise, we need to probe the consistency control table with the required key values at run-time. For example, for Qa with $key = c_nationkey, we have two scenarios:

In the first scenario, we have to first calculate which nations are in the results, and then check if they all appear in the consistency control table C_CsCT (A11a). A more precise guard (A11b) only checks nations with more than one customer, by adding the COUNT(*)>1 condition. Checking like A11a, A11b and A12 is called **assured consistency checking** in that it checks if the required consistency groups are part of the guaranteed cache regions.

In the second scenario, a redundant equality predicate on c_nationkey is included in the query, allowing us to simply check if the required nations are in C_CsCT (A12). It eliminates the need to examine the data for consistency checking.

Condition D: Can each required consistency group be covered by a collection of cache regions.

If Yes, we have the opportunity to do ad-hoc consistency checking. For Qa with $key = Ø, we check if all the required customers are in the same ad-hoc cache region (S11). Such checking (e.g., S11, S12 and S21, S22 from Figure 5.5) is called **ad-hoc consistency checking**.

If $key=c_nationkey and suppose we don't have C_CsCT, we need to check each group (S12).

```
A11a, A11b:   SELECT    1    WHERE  NOT EXISTS (
              SELECT  1        FROM  CustCopy
              WHERE   c_custkey IN $custSet
              GROUP BY c_nationkey
              HAVING [COUNT(*) > 1 AND] c_nationkey NOT IN
                  (SELECT nkey FROM C_CsCT) )

A12:   SELECT  1        WHERE   |$nationSet| =  (
              SELECT    COUNT(*)       FROM     C_CsCT
              WHERE     nkey IN $nationSet  )

S11:   SELECT  1        WHERE   1 = (
              SELECT    COUNT(DISTINCT rid)   FROM C_PCT
              WHERE     ckey IN $custSet )

S12:   SELECT    1      WHERE   1 = ALL (
              SELECT    COUNT(DISTINCT rid)   FROMC_PCT, CustCopy
              WHERE     c_custkey IN $custSet AND key=c_custkey
              GROUP BY c_nationkey    )

S21:   SELECT    1      FROM (
              SELECT    COUNT (DISTINCT rid1) AS count1,
                        SUM (ABS(rid1-rid2)) AS count2
              FROM (
                  SELECT    A.rid AS rid1, B.rid AS rid2)
                  FROM      C_PCT A, O_PCT B
                  WHERE     A.ckey IN $custSet AND A.ckey = B.ckey) )
                        AS FinalCount
              WHERE     count1 = 1 AND count2 = 0   )

S22:   SELECT    1      WHERE   NOT EXISTS (
              SELECT    1      FROM (
                  SELECT    c_custkey, c_nationkey,
                            A.rid AS rid1, B.rid AS rid2
                  FROM      C_PCT A, O_PCT B, CustCopy C
                  WHERE     A.ckey IN $custSet AND
                            A.ckey = c_custkey AND c_custkey = B.ckey
                        ) AS FinalCount
              GROUP BY c_nationkey
              HAVING  MIN(rid1) <> MAX(rid1 OR MIN(rid2) <> MAX(rid2)
                      OR MIN(rid1) <> MIN(rid2))
```

Figure 5.5 Consistency guard examples

	Local			Remote		
	Qa	Qb	Qc	Qa	Qb	Qc
Cost (ms)	.078	.08	1.17	.01	.19	1.13
Cost (%)	16.56	14.00	<2	<1	<2	<1
# Rows	1	6	5975	1	6	5975
Base query (ms)	0.45	0.57	67.99	5.54	11.72	70.77

Table 5.1 Simple consistency guard overhead

Experiment 1 is designed to measure the overhead of the simple consistency guards supported in our current framework. We chose to support only run-time consistency guards that 1) do not require touching the data in a view; 2) only require probing a single control table. We fixed the guards and measured the overhead for: Qa and Qb with $custSet = (1); Qc with $nationSet = (1). The consistency guard for Qa and Qb is S11 and the one for Qc is A12.

The results are shown in Table 5.1, where the last row is the execution time for the queries without run-time consistency checking. As expected, in both the local and remote case, the absolute cost remains roughly the same, the relative cost decreases as the query execution time increases. The overhead for remote execution is small (< 2%). In the local case, the overhead for Qc (returning ~6000 rows) is less than 2%. Although the absolute overhead for Qa and Qb is small (<0.1ms), since the queries are inexpensive (returning 1 and 6 rows respectively), the relative overhead is ~15%.

	Local					Remote				
	A11a	A11b	A12	S11	S12	A11a	A12	A12	S11	S12
Cost (ms)	.31	.12	.084	.29	.35	.33	.27	.13	.41	.48
Cost (%)	62.85	23.77	16.98	58.32	71.41	6.06	4.95	2.33	7.48	8.79
Qa (ms)	.49	.49	.49	.49	.49	5.44	5.44	5.44	5.44	5.44

Table 5.2 Single-table case consistency guard overhead

In experiment 2, we used query Qa with $custSet = (2, 12), which returns 2 rows; and compared the overhead of different types of consistency guards that involve one control table. The results are shown in Table 5.2, where the last row is the execution time for the queries without run-time checking.

For local execution, if the consistency guard has to touch the data of the view (A11a, A11b and S12), the overhead surges to ~70% for S12, because we literally execute the local query twice. A11a and b show the benefit of being more precise: the “sloppy” guard in A11a incurs 63% overhead, while the overhead of the more precise guard (A11b) is only 24%, because it is less likely to touch CustCopy. The simple guard A12 incurs the smallest overhead (~17%).

5.4.4.2 Multi-Table Case

Different from Qa, the required consistency group in Qb has objects from different views. In this case, we first check:

Condition E: Do they have the same consistency root?

	Local		Remote	
	S21	S22	S21	S22
Cost (ms)	.90	.83	1.00	.98
Cost (%)	155.83	143.82	24.82	24.36
Qb (ms)	.58	.58	4.02	4.02

Table 5.3 Multi-table case consistency guard overhead

If Yes, then the consistency guard generation reduces to the single table case, because the guaranteed cache regions are decided by the consistency root. Otherwise, we have to perform ad-hoc checking involving joins of presence control tables. There are two cases.

Case 1: \$key = Ø. We check if all the required presence control-keys point to the same cache region (S21).

Case 2: \$key = c_nationkey. We first group the required rows by c_nationkey, and check for each group if 1) all the customers are from the same region; and 2) all the orders are from the same region as the customers (S22).

In Experiment 3, we use query Qb with \$custSet = (2, 12), which returns 7 rows, and measure the overhead of consistency guards that involve multiple control tables. The results are shown in Table 5.3, where the last row is the execution time for Qb without run-time checking. Note that the execution time for the remote execution of Qb in this setting is different than that in the first experiment (Table 5.1). The reason is that different plans were generated and used for those two settings. This difference does not affect the results of this study. Guards S21 and S22 involve not only accessing the data, but also performing joins.

Such complicated checking incurs huge overhead in the local execution case (~150%). Note that if CustCopy and OrderCopy are consistency-wise correlated, then the overhead (refer to the single-table case) reduces dramatically.

It is worth pointing out that in all experiments, even for complicated consistency guards, the overhead of remote execution is relatively small (<10% for single-table case, <25% for multi-table case). It raises an interesting point: even if a guard is less likely to be successful, it might still be preferable to do the check than simply use a remote plan. Thus the cost-model should bias in favor of plans with consistency checking instead of remote plans.

Chapter 6

Quality-Aware Database Caching Performance Modeling: Alternatives and Implications

6.1 Introduction

Chapter 1 - Chapter 4 has presented a novel database caching model where queries are allowed to express their data quality requirements; the cache may choose to maintain different levels of local data quality; and the caching DBMS provides quality guarantees for query results by shifting between using local data and remote data from the master. The use of replication is not free, because of the potential update propagation cost. In such a complex system, the performance improvements provided by caching "good enough" copies, if any, will be determined by many factors, including query workload, update workload and the system configurations. The goal of this chapter is to explore the system design space, quantify the impact of alternatives, and offer insights into performance tradeoffs.

It is worth emphasizing what we are NOT set out to do. First, we do NOT intend to investigate the adaptive caching aspect of this problem, an example of which is cache replacement or admission policies. Rather, we examine the influence of design alternatives under the full replication assumption (i.e., a cache has all the objects of the master). Second, it is not our purpose to predict the performance of some actual system. Rather, we try to exemplify under what circumstances gains can be achieved for what design and what the magnitude of these gains might be.

We begin by establishing a performance evaluation framework based on a simple but realistic model. Our model is an extension of a database management system model developed by Agrawal et al. [ACL87], which we refer to as the *Agrawal model* for convenience. The Agrawal model captures the main elements of a database environment, including both *users* (i.e., terminals, the sources of transactions) and *physical resources* for storing and processing the data (i.e., disks and CPUs), in addition to the characteristics of the workload and the database. We extend the single-site model to a cache-master configuration, capturing the interaction between the cache and the master. In addition, we refine the single-site model to reflect the characteristics of cache organization and maintenance.

Based on this framework, we examine the effects of alternative assumptions and system configurations. In particular, we critically examine the common assumption that performance improves as the number of caches increases. The use of a cache is not free; workload offloading is achieved at the cost of update propagation. We show under what conditions the performance can be improved by adding more caches, and when the overhead of maintaining a cache exceeds the benefit it can bring.

Allowing queries to express their C&C requirements gives the cache freedom in maintenance, because if local data does not meet the specified data quality requirements, the cache can simply shift the query to the master, thus providing desired data quality guarantees. Obviously, the tradeoff is between workload offloading and maintenance overhead. We quantify the performance differences with different tradeoffs.

In addition to these experiments quantifying the effectiveness in relaxing C&C requirements and local data quality, we examine the impact of the push maintenance and the pull maintenance. The push maintenance uses incremental update propagation; but can only be applied to view-level cache regions, thus not providing support for finer granularity caching. In contrast, the pull mode is more flexible, supporting finer-grained (hence smaller) cache regions, but requires recalculation. By experimentally showing which alternatives work best for what workload, we provide insights into choosing appropriate system configurations.

The rest of the chapter is organized as follows. Section 6.2 describes the push and pull maintenance. We implement the model of cache-master architecture using the CSIM discrete event simulation package [Sch86, Mesquite] for our performance studies, which is based on a closed queuing model. The structure and characteristics of our model are described in Section 6.3. Section 6.4 discusses the performance metrics and statistical methods used for the experiments, and how a number of our parameter values were chosen. We present the experimental results and analysis in Section 6.5 and conclude in Section 6.6.

6.2 Push vs. Pull Maintenance

In Chapter 4 we briefly described the push maintenance for view level granularity cache regions, where Microsoft SQL Server’s transactional replication feature [Ise01, Hen04] is used to propagate changes to the cache. With this model, the master database can specify a set of materialized views as *publications*. A cache can subscribe to views from one or more publications. Each subscriber corresponds to a distribution agent (an OS process). Changes to a publication are recorded to a separate database, called distribution database using a log sniffing technique. Periodically, a dispatch agent wakes up and propagates relevant changes to its subscriber. After all the subscribers of the publication have received the changes, the records are deleted from the distribution database.

Such maintenance is efficient, since the cache is updated incrementally. However, it imposes two restrictions on the cache organization: 1) only view level cache regions are supported; 2) only limited types of views, i.e., selection and projection views are supported.

In comparison, the pull maintenance we introduced in Chapter 4 employs a re-computation approach. For a cache region, the cache periodically refreshes it by sending queries to the master. The newly retrieved data calculated from the current state of the master is then used to replace the stale data in the cache. This model offers maximal flexibility in cache region organization, since we can choose to refresh any part of a view by sending a remote query. Although re-computation can be expensive, this approach can be useful in cases where skewed data quality requirements to a view are observed.

6.3 Performance Model

Our model is built on top of the Agrawal model, which is an extended version of the model used in [CS84], which in turn has its origins in the models of [RS77, RS79a]. Although the

Agrawal model is developed to evaluate the performance of a set of locking protocols, since it models a database system in a fairly detailed and complete fashion, we were able to adopt it for our own purpose.

We begin by explaining the Agrawal model in Section 6.3.1. For a cache-master configuration, we extend this model in three aspects. Firstly, we extend the single-site database model to the cache-master configuration. This is achieved by modeling interactions between the cache and the master during query processing as well as during cache maintenance (*Section 6.3.2*). Secondly, we capture C&C related characteristics for a cache-master configuration, for instance, workload with C&C constraints, cache region concept, and C&C-aware transaction processing at the cache (*Section 6.3.3*). Finally, we refine the model to differentiate sequential access from random access, which is critical for our study, because “pull” maintenance tends to be sequential, and “push” maintenance random. (*Section 6.2*).

6.3.1 Single Site Database Management Systems

In this section, we describe the Agrawal model for single site database management systems. There are three main parts to the performance model: a database system model, a user model, and a transaction model. The *database system model* captures the relevant characteristics of the system’s hardware and software, including the physical resources (CPUs and disks) and their associated schedulers, the characteristics of the database (e.g., its size and granularity), the load control mechanism for controlling the number of active transactions, and the concurrency control algorithm. The *user model* captures the arrival process for users, assuming a closed system with terminals, batch-style (non-interactive) in nature. Finally, the *transaction model* captures the reference behavior and processing requirements of the transactions in the workload. A transaction submitted at the master can be described via two characteristic strings with C&C components. There is a logical reference string, which contains concurrency control level read and write requests; and a physical reference string, which contains requests for accesses to physical items on disk and the associated CPU processing time for each item accessed. In addition, if there is more than one class of transactions in the workload, the transaction model must specify the mix of transaction classes.

The closed queuing model of a single-site database system is shown in Figure 6.1. There are a fixed number of terminals from which transactions originate. There is a limit to the number of transactions allowed to be active at any time in the system, depicted by the multiprogramming level *mpl*. A transaction is considered active if it is either receiving service or waiting for service inside the database system. When a new transaction is generated, if the system already has a full set of active transactions, it enters the *ready queue*, where it waits for a currently active transaction to complete or abort. (Transactions in the ready queue are not considered active.) The transaction then enters the *cc queue* (concurrency control queue) and makes the first of its lock requests. If the lock request is granted, the transaction proceeds to the *object queue* and accesses its first object. It is assumed for modeling convenience that a transaction performs all of its reads before performing any writes.

If the result of a lock request is that the transaction must block, it enters the *blocked queue* until it is once again able to proceed. If a request leads to a decision to restart the transaction due to deadlock, it goes to the back of the ready queue, after a randomly determined restart delay period of the observed response time. It then begins making all of the same concurrency control requests and object accesses over again. Eventually, the transaction may complete and the concurrency control algorithm may choose to commit the

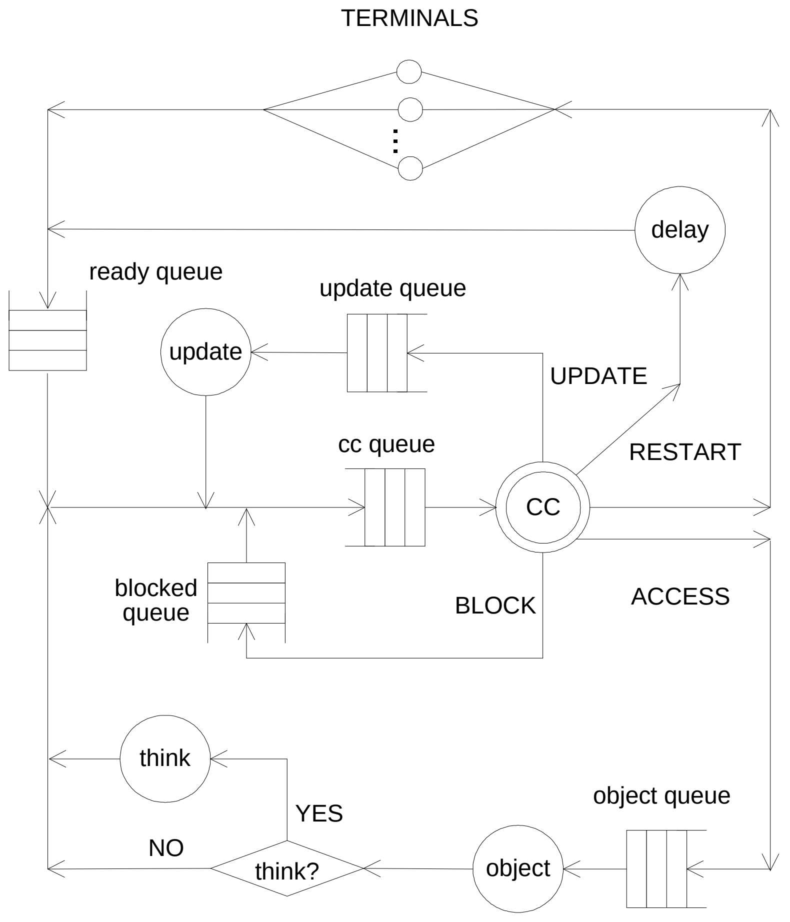


Figure 6.1 Logical queuing model (From [ACL87])

transaction. If the transaction is read-only, it is finished. If it has written one or more objects during its execution however, it first enters the *update queue* and writes its deferred updates into the database. Locks are released together at the end-of-transaction (after the deferred updates have been performed). Wait queues for locks and a wait-for graph are maintained by a *resource manager*.

Underlying the logical model of Figure 6.1 are two physical resources, the CPU and the I/O (i.e., disk) resources. The amounts of CPU and I/O time per logical service are specified as model parameters. The physical queuing model is depicted in Figure 6.2, and Table 6.1 summarizes the associated model parameters for the whole system. As shown, the physical

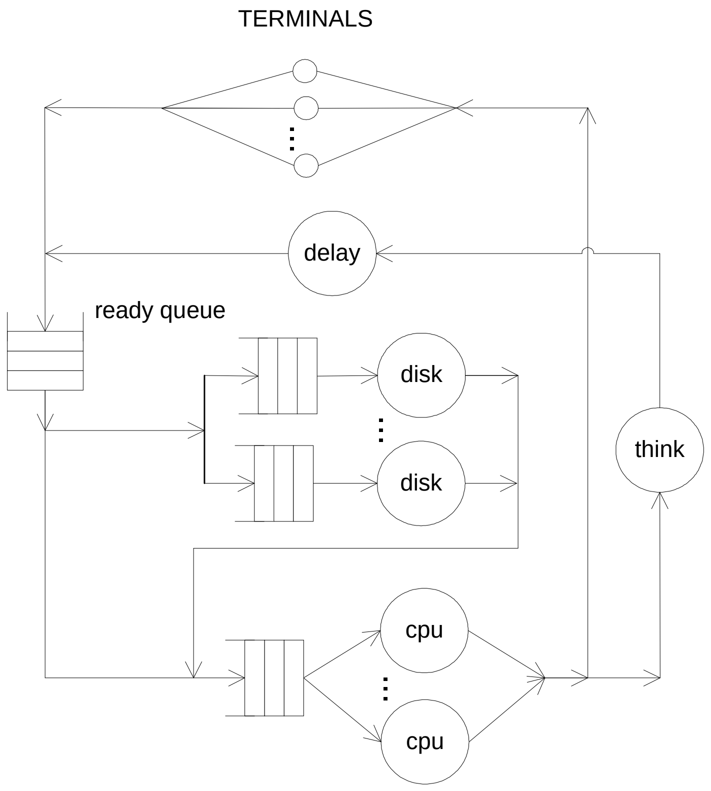


Figure 6.2 Physical queuing model (From [ACL87])

model is a collection of terminals, multiple CPU servers, and multiple I/O servers. The delay paths for the think and restart delays are also reflected in the physical queuing model. Model parameters specify the number of CPU servers, the number of I/O servers, and the number of terminals for the model. When a transaction needs CPU service, it is assigned a free CPU server; otherwise the transaction waits until one becomes free. Thus, the CPU servers may be thought of as a pool of servers, all identical and serving one global CPU queue. Requests in the CPU queue are serviced FCFS (first come, first served). The I/O model is a probabilistic model of a database that is spread out across all of the disks. There is a queue associated with each of the I/O servers. When a transaction needs service, it chooses a disk (at random, with all disks being equally likely) and waits in an I/O queue associated with the selected disk. The service discipline for the I/O requests is also FCFS.

The parameters *obj_io* and *obj_cpu* are the amounts of I/O and CPU time associated with reading or writing an object. Reading an object takes resource time equal to *obj_io* followed by *obj_cpu*. Writing an object takes resources equal to *obj_cpu* at the time of the write request and obj_io at deferred update time, since it is assumed that transactions maintain deferred update lists in buffers in main memory. For simplicity, these parameters represent constant service time requirements rather than stochastic ones. The *ext_think_time* parameter is the mean time delay between the completion of a transaction and the initiation of a new transaction from a terminal. We assume it is exponentially distributed.

Parameter	Meaning
db_size	Number of objects in database
num_terms_total	Total number of terminals
num_terms_cache	Number of terminals at a cache
mpl	Multiprogramming level
max_size	Size of largest transaction
min_size	Size of smallest transaction
write_prob	Pr (write X | read X)
read_only_percentage	Percentage of read-only transactions
ext_think_time	Mean transaction think time
obj_io	Disk time for accessing an object
obj_io_seek	Disk seeking time
obj_io_transfer	Disk transfer time for an object
obj_cpu	CPU time for accessing an object
num_cpus	Number of CPUs
num_disks	Number of disks
num_caches	Number of caches
network_delay_query	Network delay for sending a query
network_delay_transfer	Network delay for sending an object
log_sniffing_fixed_cpu	Fixed part of CPU time for log sniffing a transaction
log_sniffing_unit_cpu	Unit CPU time for log sniffing a write action
log_sniffing_fixed_disk	Fixed part of Disk time for log sniffing a transaction
log_sniffing_unit_disk	Unit Disk time for log sniffing a write action
distribution_fixed_cpu	Fixed part of CPU time for distributing updates
distribution_unit_cpu	Unit CPU time for distributing a write action
distribution_fixed_disk	Fixed part of Disk time for distributing updates
distribution_unit_disk	Unit disk time for distributing a write action
seq_prob_copier	Pr (copier reads are sequential)
seq_prob_refresh	Pr (copier writes are sequential)
num_regions	Number of cache regions at each cache
max_num_classes	Maximal number consistency classes per Xact
min_num_classes	Minimal number of consistency classes per Xact
num_classes	Number of consistency classes of the database
refresh_interval	Refresh interval
currency_bound	Currency bound

Table 6.1 Model parameters

A transaction is modeled according to the number of objects that it reads and writes. The number of objects read by a transaction is chosen from a uniform distribution between *min_size* and *max_size* (inclusive). These objects are randomly chosen (without replacement) from all of the objects in the database. The probability that an object read by a transaction will also be written is determined by the parameter *write_prob*. The size of the database is *db_size*.

6.3.2 Cache-Master Configuration

In this section, we describe the extension to the Agrawal model that reflects the C&C-aware features for a cache-master configuration. The modified logical queuing model is shown in Figure 6.3.

6.3.2.1 Cache Region Concept

A cache has the same set of objects as the master. Objects in a cache are logically partitioned into *cache regions*. All the objects from the same cache region are mutually consistent at all times. Currency information is managed, and update propagation is carried out at the granularity of cache region. For simplicity, we assume equal size cache regions. The number of regions in a cache is controlled by model parameter *num_regions*. Objects are mapped to regions by a default mapping: object *i* belongs to cache region ((*i*num_regions)*/*db_size*). Each cache region is associated with the following metadata:

1) Cache region ID: a unique identifier of the region.

2) Synchronization timestamp: the timestamp of the last synchronization point, which is used for approximating the currency of this cache region. Suppose the timestamp is t_1, and current time is t, then the currency of the region is less than $(t – t_1)$.
3) Refresh interval: the refresh interval for this cache region.

6.3.2.2 Transaction Model

In our setting, there are transactions generated by the terminals as well as transactions generated by the system (which will be explained shortly). To differentiate, we call them *user transactions* and *system transactions* respectively. In comparison to the Agrawal model, user transactions have richer semantics in our setting. Besides the basic characteristics, a user transaction also includes C&C requirements. To reflect that, we partition the objects in a user

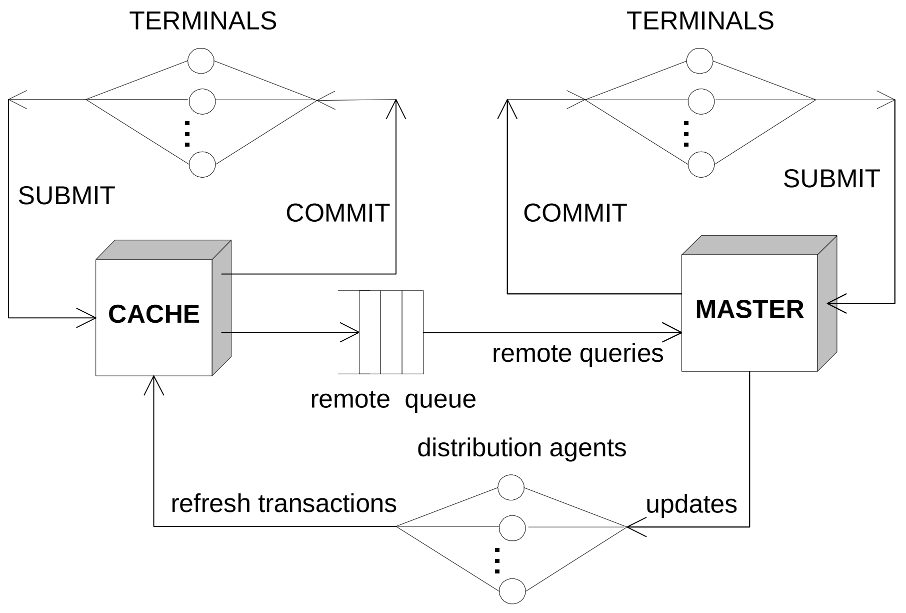


Figure 6.3 Queuing model for a cache-master configuration

transaction into consistency classes, each with a currency bound. Recall from Chapter 4 that the C&C constraints mean that all the objects in the same consistency class have to be mutually consistent, and no older than the currency bound.

For simplicity, we assume the cache is partitioned into equal length consistency classes. The number of classes in a cache is controlled by model parameter *num_classes*. Objects are mapped to consistency classes by a default mapping: object *i* belongs to class ((*i*num_classes)*/*db_size*). We also assume a consistency class can be mapped to one and only one cache region. That is, the cache always satisfies the consistency requirement of the user transactions. The number of consistency classes a user transaction accesses is uniformly distributed between *min_num_classes* and *max_num_classes* (inclusive). The consistency classes are randomly chosen (without replacement) from among all consistency classes in the cache. Assuming n classes are accessed by a user transaction, for a chosen class, the number of objects the user transaction reads is uniformly distributed between *min_size*/n and *max_size*/n. Those objects are chosen (without replacement) from among all of the objects contained in this region. Thus the minimal and maximal sizes of the read set of a transaction continue to be *min_size* and *max_size*. Similar to the Agrawal model, the probability that an object read by a transaction will also be written is determined by the parameter *write_prob*. Unless otherwise noted, the currency bound associated with a consistency class is *currency_Bound*, and the refresh interval for a cache region is *refresh_interval*.

6.3.2.3 Cache Maintenance

In our cache model, all read-write user transactions are processed at the master and we propagate changes to the cache periodically at the granularity of a cache region. To reflect that, we associate each cache region with a *distribution agent* that wakes up periodically according to the refresh interval of this cache region and propagates changes from the master to the cache.

In the "pull" model, each time a *distribution agent* wakes up, it first generates a remote transaction that reads all the objects of this cache region, which we call a *copier transaction*. The agent enters the *remote queue* and waits for the return of the remote transaction. Then it generates a transaction consisting of writes to all the objects of the cache region, called a *refresh transaction*.

In the "push" model, the master maintains a *delta queue* for each cache region, which stores the delta changes relevant to that cache region. After a read-write transaction at the master commits, we enter each of its writes into its corresponding *delta queues*. The transactional boundary and commit order for the writes in each *delta queue* are preserved. In SQL Server's Transactional Replication, such delta queues are produced using a log sniffing technique. To model the cost of log sniffing, each time we put a write into the *delta queues*, we charge a CPU cost followed by a disk cost. The CPU cost has an amortized fixed component *logSniffing_fixed_cpu* and a unit component for each object written out *logSniffing_unit_cpu*. Likewise, the disk cost has two components: *logSniffing_fixed_disk* and *logSniffing_unit_disk*.

When a *distribution agent* wakes up, it empties the corresponding *delta queue*, ships all the actions to the cache, and executes a batch of refresh transactions one by one. The distribution cost is charged similar to the sniffing cost with model parameters *distribution_fixed_cpu*, *distribution_unit_cpu*, *distribution_fixed_disk*, and *distribution_unit_disk*. Each refresh transaction consists of all writes from the same user transaction, and the refresh transactions are executed in the commit order of their corresponding user transactions.

Note that in our model, the distribution agents consume the resources (i.e., CPUs and disks) of the master site. In reality the distribution database can sit on a separate machine, which is similar to adding more resources dedicated to the distribution agents to the master. This is valid for our study since our focus is to examine the performance with a fixed number of resources assigned to the master.

6.3.2.4 Keeping Track of Currency

In the "pull" model, after each refresh, the timestamp of the corresponding *copier transaction* is used as the current timestamp of the cache region.

In the "push" model, we record the timestamp of a user transaction along with its writes in a *delta queue*. After each refresh transaction, we use the timestamp of the corresponding user transaction as the current timestamp of the cache region. Supposing the system clock is t_1 when a distribution agent wakes up, we assume that all the changes up to t_1 have already been added to its delta queue. Thus, after the whole batch of refresh transactions finishes, we set the timestamp of the cache region to t_1. Note that this timestamp is set to t_1 when the agent wakes up even in case the delta queue is empty (i.e., no changes till t_1).

6.3.2.5 Transaction Processing

There are two types of transactions in our system: user transactions generated by the terminals and system transactions generated by the *distribution agents*. We give system transactions higher priority over user transactions: as soon as a system transaction is generated, it becomes active immediately. We achieve that by bypassing the ready queue for system transactions. That is, the multiprogramming level only restricts user transactions, not the system transactions.

The master executes transactions in the same way as in the Agrawal model; it simply ignores any C&C components of a query, since all C&C requirements will be satisfied at the master.

A cache differentiates three types of transactions and handles them differently: read-write transactions, read-only transactions, and refresh transactions. While the first two are user transactions, the last one is of the system transactions type. Read-write transactions are directly forwarded to the master; C&C checking is enforced for read-only transactions; and refresh transactions are processed in the same manner as in the single-site case. In what follows, we describe each type of user transactions in more detail.

For a read-write transaction A, the moment it becomes active, we send a separate transaction A’ (identical to A) to the master, and put A into the remote queue. After A’ finishes and returns from the master, A simply commits.

For a read-only transaction B, if B has mismatched consistency classes, we treat it the same way as a read-write transaction. Otherwise, we model two alternatives in currency checking: cache region level and query level. For the former, we process each consistency class specified in B in turn. Before requesting any read lock for a cache region, we first check the timestamp of the region against the currency bound specified in the query. If the checking succeeds, reads to this consistency class will be performed locally; otherwise, all the reads in the same consistency class are sent in a separate transaction B’ to the master and B enters the *remote queue*. B will proceed again only if B’ finishes and returns from the master. It commits when all the reads are finished.

6.3.2.6 Network Cost

In addition to CPU and disk, we model a third physical resource: the network. Each remote query to the master uses a roundtrip of the network resource. Assuming infinite bandwidth, when a transaction needs the network resource, it immediately gets it. We assume the network delay for sending a remote query is the same for all queries, which is specified as model parameter *network_delay_query*. The network transfer time per object is specified as model parameter *network_delay_obj*. Suppose the number of reads in a remote query is n, then the total round-trip delay of the remote query is calculated by:

network_delay_query + n * *network_delay_obj*

6.3.3 Model Refinement

In this section, we refine the model to capture characteristics that are critical for our purpose, but not for the Agrawal model. In particular, Agrawal et al. did not use workloads containing several classes of transactions; neither did they model sequential vs. random accesses, since neither is necessary for their specific goal. Both characteristics are critical in our model given the asymmetric nature of the cache and the master in transaction handling, and the significant difference in access patterns using the pull or push maintenance alternatives.

6.3.3.1 Locking

For the master, strict two-phase locking is used to guarantee serializability. For the cache, since it only guarantees consistency within each cache region, we apply strict two-phase locking for the duration of each region access. That is, all the locks on objects in a region are released together at the point when accesses to the region finish.

We check for deadlock upon each lock request. If a lock request is to cause a deadlock, we abort the transaction which makes the request. To avoid it causing deadlocks repeatedly, before it restarts, we hold the aborted transaction for an exponential delay with a mean equal to the running average of the transaction response time — that is, the duration of the delay is *adaptive*, depending on the observed average response time.

6.3.3.2 Sequential vs. Random Access

With a pull model, reads/writes will most likely be sequential, since a copier needs to read from the master all the objects in the whole cache region, and write them back into the cache. In contrast, with a push model, the writes to the cache are more likely to be random, since only updates are propagated to the cache. We differentiate these two access patterns in our model in the following way.

Firstly, instead of assuming that objects are spread among the disks uniformly, we assume that cache regions are spread among the disks uniformly. For all experiments, we keep a fixed mapping between cache regions and disks. Suppose there are N disks, then the i^{th} cache region belongs to disk (i mod N). If the number of regions is less than the disks, we bypass this mapping and apply the disk assignment approach described in Section 6.3.1

Secondly, instead of having a single parameter *obj_io* to model the IO cost, we use two model parameters *obj_io_seek* and *obj_io_ transfer*. The IO cost of sequentially reading m objects is (*seeking_cost* + m * *transfer_cost*).

In addition, we adjust the locking protocol as follows. If a transaction reads/writes a set of objects sequentially, it has to lock them all before it accesses them, because otherwise the sequential access pattern cannot be guaranteed.

Accordingly, in our *transaction model,* we add one more characteristic to a system transaction, a flag indicating whether the reads in this transaction can be proceeded sequentially. The probability that the reads/writes of a system transaction are sequential is represented by a model parameter *seq_prob_copier*/*seq_ prob_refresh*. Intuitively, these two probabilities are much greater for the pull maintenance than for the "push" one.

We could have added the sequential vs. random access characteristics to user transactions in a similar way. We chose not to because the focus of this study is not the absolute performance of the system, rather, the comparison between different configurations. The behavior of user transactions does not change under those configurations.

6.4 General Experiment Information

The remainder of this chapter presents results from a number of experiments designed to investigate system performance with the alternative configurations discussed in Section 6.2. In this section we look at the performance metrics and statistical methods used in the rest of this study, and how we chose many of the parameter settings used in the experiments.

6.4.1 Performance Metrics

The primary performance metric used throughout the study is the user transaction throughput rate, which is the number of user transactions completed per second. We used Condor [Condor] to run all our experiments. Each simulation was run for 20 batches with a large batch time to produce sufficiently tight 90 percent confidence intervals. The actual batch time was 100,000 seconds (simulation time). Our throughput confidence intervals were typically in the range of plus or minus a few percentage points of the mean value, more than sufficient for our purposes. We omit confidence interval information from our graphs for clarity, but we discuss only the statistically significant performance differences when summarizing our results.

In analyzing the experiment results, we divide the system throughput into three categories, and measure them separately:

1) Throughput for read-only transactions submitted to the master.
2) Throughput for read-only transactions submitted to the caches. Parts of the reads might actually be executed at the master depending on the result of currency checking.
3) Throughput for read-write transactions. All read-write transactions are executed at the master, regardless where they are submitted. This metric helps to understand the number of updates that need to be propagated per synchronization.

We also measure the throughput for system transactions. Again, we measure it separately for copier transactions and refresh transactions.

Response times, expressed in seconds, are also given in some cases. These are measured as the difference between when a transaction is generated and when the transaction returns following its successful completion, including any time spent waiting in the ready queue, time spent before (and while) being restarted, and so forth. We measure the response time separately for each category in the same way as for throughput.

To understand the effectiveness of replication, we measure the local workload ratio, which is the ratio of the number of reads completed at the caches to the total number of reads submitted to the cache.

Several additional performance-related metrics are used in analyzing the results of our experiments. We measure the average batch size (measured in actions) per synchronization, which determines the minimal time needed for executing the batch of transactions, hence indicates the effectiveness of synchronization.

In monitoring the effects of data contention, two conflict-related metrics are employed. The first metric is the blocking ratio, which gives the average number of times that a transaction has to block per commit (computed as the ratio of the number of transaction-blocking events to the number of transaction commits). The other conflict-related metric is the restart ratio, which gives the average number of times that a transaction has to restart per commit (computed similarly). We measure the conflict for both the master and the caches. Another set of metrics used in our analysis is the workload shift percentage measured at the object level, which gives the ratio of reads that pass currency checking to the total reads generated from a cache.

The last set of metrics used in our analysis is the average disk utilization, which gives the fraction of time during which a disk is busy. Again, we measure it both for the master and for the caches. Note that disk utilization is used instead of CPU utilization because the disks turn out to be the bottleneck resource with our parameter settings (discussed next).

6.4.2 Parameter Settings

Table 6.2 gives the values of the simulation parameters that all of our experiments have in common (except where otherwise noted). Parameters that vary from experiment to experiment are not listed there, but will instead be given with the description of the relevant experiments.

The total number of terminals is set to 300, and the number of terminals assigned to each cache is set to 20. When we increase the number of caches from 0 to 15, the number of terminals at the master decreases from 300 to 0. This gives the full spectrum of workload origination. Following [ACL87], we use 1 CPU and 2 disks as 1 resource unit and then vary the number of resources for the master and a cache. This balances the CPUs and disks, making the utilization of these resources about equal, as opposed to being either strongly CPU bound or strongly I/O bound. We set the number of resource units at the master and a cache to 2 and 1 respectively. We did not use larger numbers of terminals or resources units, because our preliminary experiments showed that the chosen setting provides reliable results while not requiring excessive simulation time.

Parameter	Value
db_size	10,000 pages
num_terms_total	300
num_terms_cache	15
mpl	50
max_size	12-page readset (maximum)
min_size	4-page readset (minimum)
write_prob	0.25
read_only_percentage	90
ext_think_time	1 second
obj_io	35 milliseconds
obj_io_seek	30 milliseconds
obj_io_transfer	5 milliseconds
obj_cpu	15 milliseconds
num_cpus (master)	2 CPUs at the master
num_disks (master)	4 disks at the master
num_cpus (cache)	1 CPUs at a cache
num_disks (cache)	2 disks at a cache
num_caches	0, 1, 3, 5, 8, 10, 13, and15
network_delay_query	20 milliseconds
network_delay_transfer	5 milliseconds
log_sniffing_fixed_cpu	15 milliseconds
log_sniffing_unit_cpu	5 milliseconds
log_sniffing_fixed_disk	20 milliseconds
log_sniffing_unit_disk	5 milliseconds
distribution_fixed_cpu	200 milliseconds
distribution_unit_cpu	5 milliseconds
distribution_fixed_disk	200 milliseconds
distribution_unit_disk	5 milliseconds
seq_prob_copier	1
seq_prob_refresh	0 for push, 1 for pull
num_regions	1

Table 6.2 Simulation parameter setting

We set the database size to 10,000 pages, the average size of a read-only transaction to 8 reads, and the write probability is set to 0.25. Thus in average, there are 8 reads and 2 writes in a read-write transaction. The multiprogramming level is set to 50. Our preliminary experiments showed that this setting causes significant resource contention with minimal data contention (due to locking). This is desirable since we are not interested in locking performance, but rather system behavior due to limited resources. Note that the multiprogramming level is internal to the database system, which controls the number of transactions that may concurrently compete for data, CPU and I/O services (as opposed to the number of users that may be attached to the system). We could have chosen a higher number, which causes even more resource contention. However, we then would lose control of the frequency of system transactions, because of the long average response time due to extremely high resource contention.

We set low amortized log sniffing cost (roughly 1/5 of the average transaction execution time), assuming log sniffing is always sequential and does not involve much CPU cost. The amortized distribution cost is set to roughly the average transaction execution time, since it involves waking up the distribution agent, and setting up a remote connection to the cache, which is costly.

For the first three experiments, the whole cache forms the largest cache region, and the whole database forms the largest consistency class. Thus each read-only transaction only has 1 consistency class.

It is worth pointing out that there are a number of parameters that we could have varied but did not. For example, we could have varied the size of databases, transactions, distribution of transaction sizes, or the database granularity; we also could have varied the number of resource units, total number of terminals and number of terminals at a cache. For the purposes of this study, such variations were not of interest. Our goal was to see how certain basic assumptions affect the system behaviors.

6.5 Experiments and Analysis

The high level goal of the experiments is to answer the following questions:

1) the impact of writes,
2) the impact of relaxing refresh intervals,
3) the impact of relaxing currency requirements and
4) the impact of push vs. pull.

The push model incurs log sniffing overhead and distribution overhead at the master. Such overhead increases as the number of caches increases. At a certain point, the refresh cost becomes the bottleneck of the system. We address this problem in Section 6.5.1. A natural response to this problem is to relax the refresh interval and hence reduce the propagation cost. The question then is if that will help? Section 6.5.2 answers this question. Even if it does help, what if we also want to provide data quality guarantees to each query? In this case, reducing refresh interval may bring a side effect: workload being shifted back to the master. In Section 6.5.3, we study the impact of relaxation of currency requirements. The first three experiments adopt a traditional setting, where the cache is always consistent. Finally, we want to examine how the system can benefit from even more relaxed cache maintenance, i.e., at the granularity of cache regions. Push and pull maintenance have different implications in the allowed granularity. First, we expect that the use of caches is not free. Although the pull model is generally more costly, it is useful for certain cases. Section 6.5.4 sheds light on different performance tradeoffs between push and pull.

Note that for illustration purpose, we include in this section only graphs that present the main results. For graphs that help to analyze and understand the system behavior, instead, we put them into the Appendix.

6.5.1 Experiment 1: Impact of Writes

We performed two sets of simulation experiments to study the system behavior under various workloads with a spectrum of read-only percentages: 100, 95, 90, 80, 70, and 50. With a lower read-only percentage, one can expect less performance improvement when the number of caches increases, because less workload can be shifted to the caches. Recall that a cache in our framework directly forwards read-write transactions to the master. The results confirm our intuition.

In the first set of experiments, we set the refresh interval to infinity. The system throughput is shown in Figure 6.4. In general, the system throughput increases as the number of caches increases. When the read-only percentage is high (100, 95, and 90), the system throughput improves roughly 6-fold. Note that a cache is only half as powerful as the master. When the read-only percentage is low (80, 70, and 50), however, the improvement becomes marginal (less than 2-fold for 0.8, less than 50% for 0.5). The difference between different curves is purely the number of read-only transactions assigned (and executed) at the caches. This is so because 1) the currency bounds are set to infinity, hence all read-only transactions submitted to the caches can be executed locally; and 2) there is no maintenance overhead since the refresh interval is infinity. From Figure 6.4, we see that the difference between the curves increases as the number of caches increases. This is because the difference in percentage of workload executed at the cache increases with the number of caches, as obvious from the following formula:

cache workload = *num_caches* * *num_terms_cache* * (*read_only_percentage/100)*

And this is confirmed by the throughput of the master (Figure 8.3) and that of the cache (Figure 8.1), from which we can see that the difference in cache throughput component dominates the difference in system throughput. In Figure 8.3, each curve goes down because the percentage of read-write (longer) transactions increases, as shown in Figure 8.4. For the last segment, a curve with a high read-only percentage (i.e., 100, 95 and 90) drops greatly, because from the above formula, the workload on the master drops greatly. In Figure 8.1, the curves are not linear. This is because the transaction generation rate is affected by the master response time. Recall that after generating a transaction, a terminal has to wait until that transaction returns before generating a new one. For the curve with read-only percentage 95, there is a bigger leap for the last segment, which is a consequence of the greater decrease in master response time (Figure 8.2).

In Figure 6.4, we see a flat segment when the number of caches increases from 13 to 15. The reason is as follows. From our parameter settings, the total number of terminals in the system is 300 and each cache is assigned 20 terminals. 13 caches leave 40 terminals at the master, while 15 caches leave none. Since the master is twice as powerful as the cache, it does not make much difference to have the 40 terminals at the master or at two more caches.

In the second set of experiments, we set the refresh interval to 0. That is, the cache is continuously refreshed, and the system throughput is shown in Figure 6.5. The behavior of the system is similar to the first case where the refresh interval is infinity, but with every curve shifted down except for the special case of a read-only percentage of 100. Cache maintenance overhead contributes to the difference. With a read-only percentage of 90 and 15 caches, the improvement reduces dramatically from 7-fold to less than 2-fold.

Last but not least, we claimed that our setting causes significant resource contention but minimal data contention. To show that this is true, we plotted the disk utilization and blocking ratio for the master and the caches respectively. We only show the results for the first setting (Figure 8.5, Figure 8.6, Figure 8.7 and Figure 8.8), but similar results are observed for the second setting. For the master, the disk utilization is above 0.95 till the number of caches reaches 13. Then the utilization drops because the master is no longer saturated. For the cache, the utilization ranges from ~0.11 to ~0.95 when the read-only percentage increases from 50 to 100, as the cache becomes saturated. The blocking ratio is less than 0.09 even for a read-only percentage of 50. That is, on average a transaction waits for less than 0.09 times before it completes. For the cache, the blocking ratio is 0. We did not show the restart ratios here since they are indistinguishable from the horizontal axis.

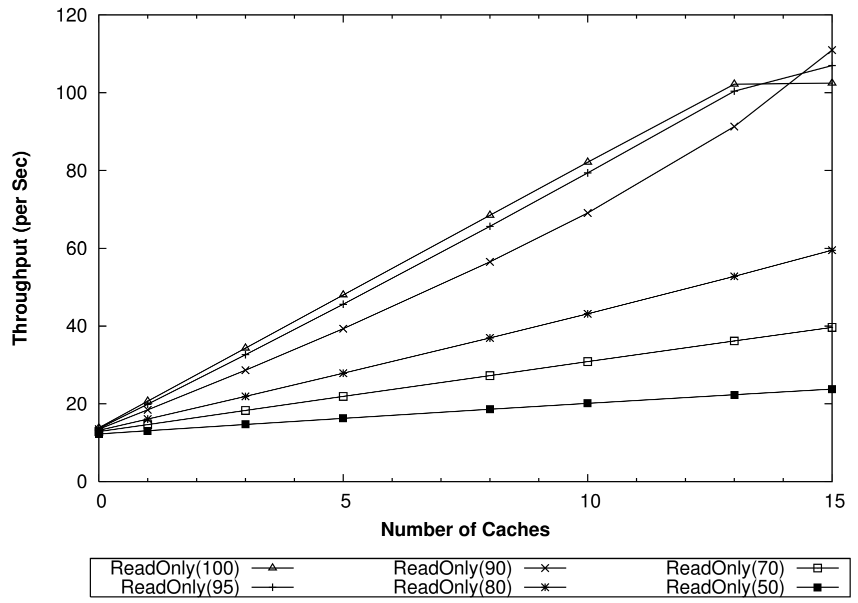


Figure 6.4 System throughput (∞ currency bound, ∞ refresh interval)

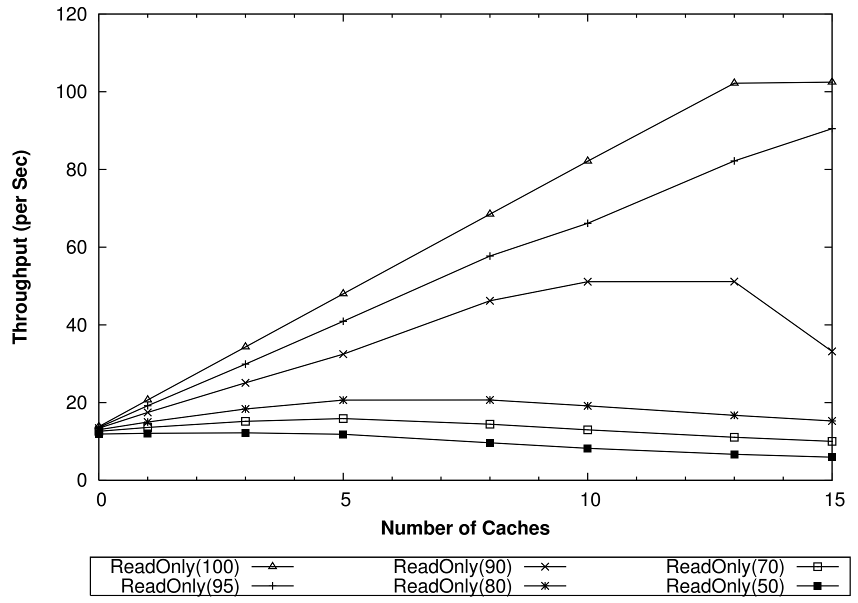


Figure 6.5 System throughput (∞ currency bound, 0 refresh interval)

6.5.2 Experiment 2: Impact of Relaxing Refresh Interval

The first set of experiments showed that the cache maintenance overhead may jeopardize the performance improvement we get by using more caches, even for a fairly high read-only percentage of 90 when we refresh the cache continuously. In this set of experiments, we examine the difference in system performance when we reduce the refresh intervals. Intuitively, when we refresh the caches less frequently, we incur less maintenance overhead, and hence get better system performance.

We fix the workload with a read-only percentage of 90 and infinite currency bound (the same setting as the curve with read-only percentage of 90 in the previous section). The refresh intervals are set to 0, 5, 10, 50 and 100 respectively, and the result is shown in Figure 6.6. Surprisingly, there is no significant difference between these settings for most cases except when there are 15 caches. When the number of caches is 15, with a refresh interval of 5 seconds, the performance improves about 50%; larger refresh intervals improve the performance by about 80%. To better understand the results, we plotted the throughput at the master (Figure 8.9) and the caches (Figure 8.10), and throughput of read-write transactions at the cache (Figure 8.11).

The system throughput is dominated by the cache throughput, since the system is dominated by read-only transactions. The throughput of the master shows a similar trend as that the system. The master directly benefits from less frequent refreshes, because in our model the distribution costs are charged to the master for each refresh. There is no significant difference when the number of caches is less than 15. The difference is more significant when the number of caches increases, because of the increase in the number of distribution agents. The throughput of the cache shown in Figure 8.10 reflects the throughput of the read-write transactions at the master, as confirmed by the similarity (except for the scale) between Figure 8.10 and Figure 8.11, for the same reason as described in the previous section: a terminal at a cache waits for a read-write transaction to return before generating a new one.

We suspected that the reason we do not see significant changes when the number of caches is less than 15 is that in our setting, the distribution costs are rather low. To verify that, we repeated the set of experiments with a heavier charge for distribution cost (4 times for *distribution_fixed_io*) and the results (Figure 6.7) confirm our conjecture. The difference between different refresh intervals is significant in this setting. By changing the refresh interval from 0 to 100, the system performance almost doubles even with only 5 caches.

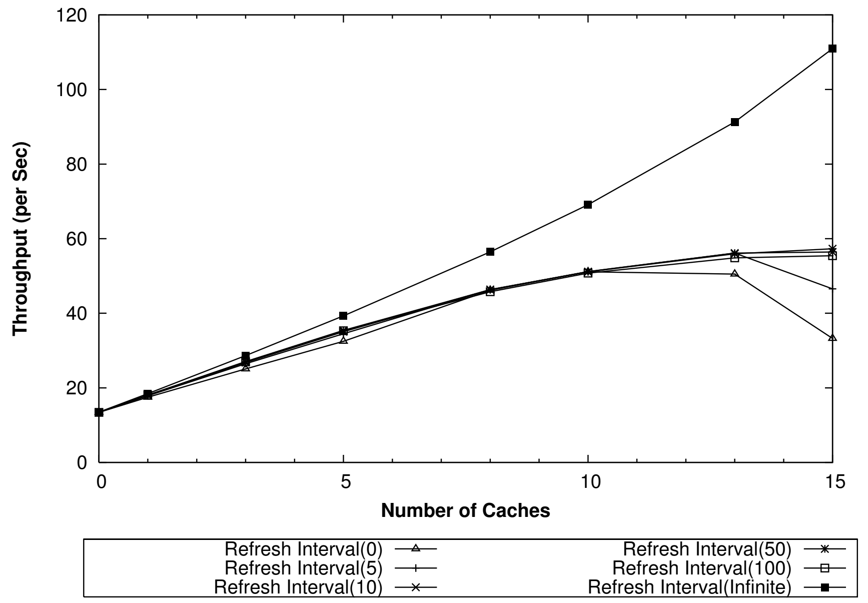


Figure 6.6 System throughput (∞ currency bound)

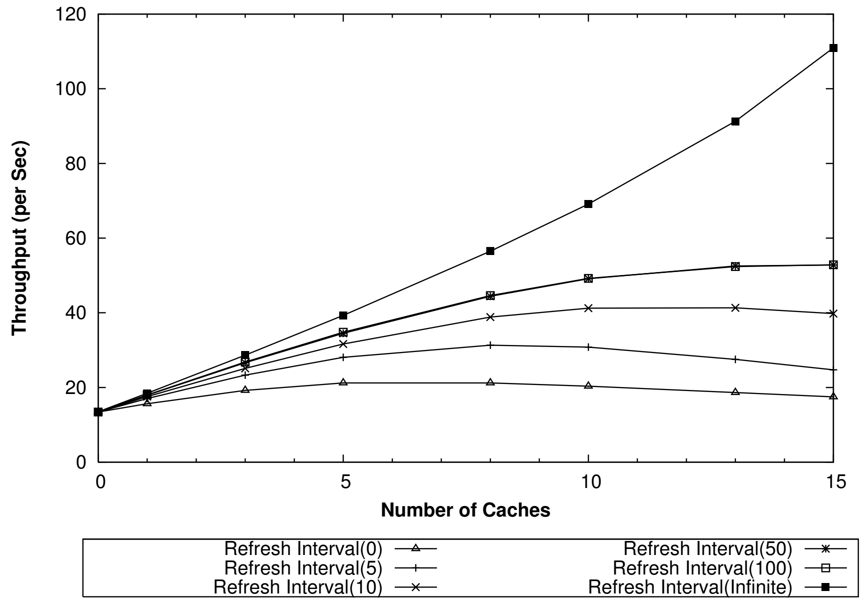


Figure 6.7 System throughput
(∞ currency bound, heavy distribution overhead)

6.5.3 Experiment 3: Impact of Refresh Interval vs. Data Quality Requirement

In the previous section, we showed that by relaxing local data quality, we can potentially improve the system performance, even when the distribution cost is relatively low. At one extreme, if we never refresh the caches, we totally avoid the maintenance overhead. However, the observed performance improvement was under the assumption that the data quality requirements from the read-only transactions are extremely relaxed (infinity). In our framework, the cache needs to satisfy data quality requirements from read-only transactions. If local data is too old for a consistency class specified in the transaction, all the reads from that class will be sent to execute at the master. Intuitively, the longer the refresh interval, the poorer the local data quality, and hence less read-only transactions can be answered from the caches. Thus, there is a trade-off between refresh overhead and workload shifted to the master. We examine this tradeoff in this section.

We conducted 4 sets of experiments, each with a different refresh interval (0, 5, 50 and infinity, measured in seconds). For each experiment, we measured the performance of workloads with different mean currency bounds (0, 5, 10, 50, 100 and infinity, measured in seconds). We first examine the results of the first set of experiments in detail, and then we compare the impact of the 4 refresh interval settings.

6.5.3.1 Experiments with Refresh Interval of zero

Figure 6.8 shows the throughput of the first set of experiments with a refresh interval of 0. The throughput of currency bound 0 and infinity serve as the lower and upper bounds respectively. When the currency bound is 0, the whole workload is executed at the master. Increasing the number of caches does not bring any benefit; worse, it imposes cache maintenance overhead and network cost to the system. Thus, the throughput decreases (from ~13 to ~8). When the currency bound is infinity, since all read-only transactions are executed at the cache, the cache can share the maximal workload from the master. (Note that there is a crossover point, which will be discussed shortly.) For most cases, relaxing currency bound significantly improves system performance. With 10 caches, for example, the system throughput is roughly 9 for currency bound 0. It increases to roughly 31, 42 and 52 when relaxing the currency bound to 5, 10 and 50 seconds respectively, giving the relative improvements of 2.4, 4.7 and 5.8-fold.

However, some characteristic of this figure is less intuitive. For the remaining settings of currency bounds, the performance first increases and then decreases. Why does it not keep increasing?

The system behavior is the result of rather complex interactions of several factors: currency bound, refresh interval, response time of copier transactions and workload at the master. (1) After each refresh, the timestamp of the cache is updated. The effectiveness of updating this timestamp depends on how often this timestamp is set, which is the refresh interval when the copier batch execution time is less than the refresh interval. Otherwise, it is the execution time, which is determined by the size of the batch, which in turn is determined by the throughput of read-write transactions at the master. (2) On the other hand, the throughput of read-write transactions at the master is influenced by the number of read-only transactions executed at the master. (3) The throughput of a cache is affected by the throughput of the master because the response time from the remote transactions it sends affects the speed of transaction generation.

We plotted the local workload ratio at the cache (Figure 6.9), throughput of the cache (Figure 8.12), and throughput of read-write transactions at the master (Figure 8.13) for analysis.

From Figure 6.9, the local workload ratio is well above 90% for all currency bounds. When the currency bound relaxes, the ratio comes even closer to and finally reaches 1. Differences between the curves are tiny. Note that given the high percentage of read-only transactions, even small changes result in a significant amount of workload shifted to the master. For example, for 15 caches, 1% workload shifted to the master equals to (300 * 0.9 * 0.01 = 2.7) read-only terminals shifted to the master.

After the number of caches reaches 13, the master is no longer saturated. Together with the fact that the master is slightly more powerful than a cache, this explains the unusual characteristics of Figure 6.8, which are shared by both Figure 8.12 and Figure 8.13. When the master is not overly burdened, the benefit of shifting more workload to new caches might be diminished by the cost of maintaining more caches. Thus the system performance may decrease.

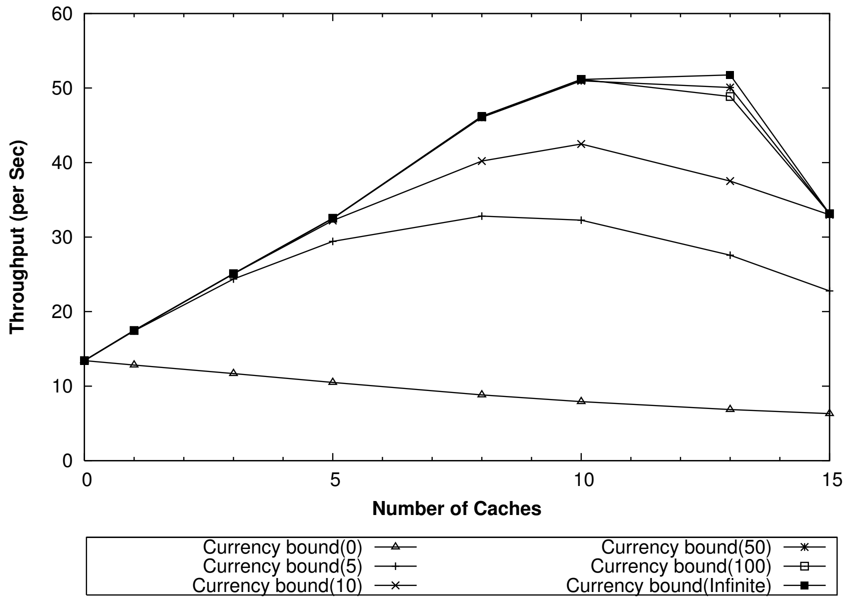


Figure 6.8 System throughput (0 refresh interval)

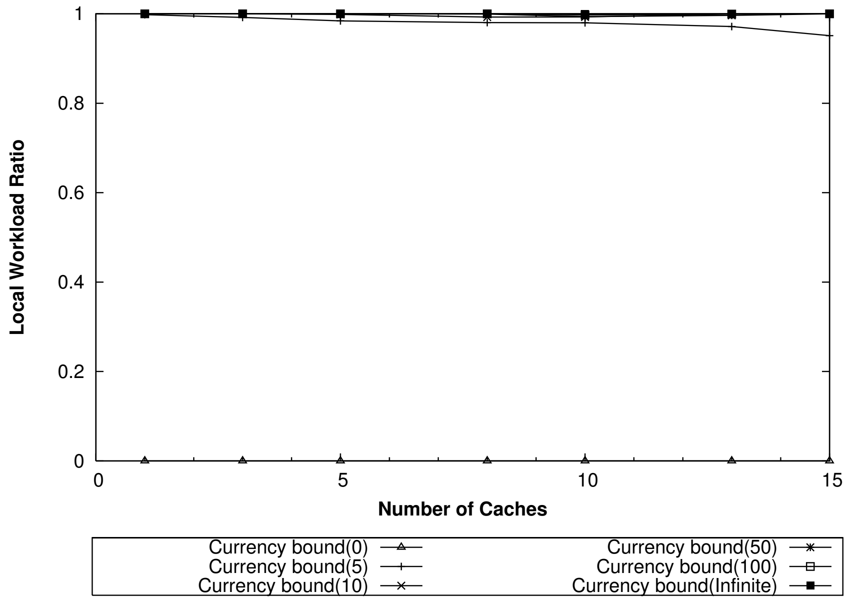


Figure 6.9 Local workload ratio for the caches (0 refresh interval)

6.5.3.2 Comparison of Different Refresh Interval Settings

The system throughput for the other sets of experiments, with refresh intervals of 5, 50 and infinity is shown in Figure 6.10, Figure 6.11 and Figure 6.12 respectively. The local workload ratio is shown in Figure 6.13, Figure 6.14 and Figure 6.15.

The figures clearly show a tradeoff in refresh interval setting w.r.t. the currency bound. To one extreme, when the refresh interval is sufficiently tight compared to the currency bound, almost all read-only transactions are executed locally at the caches. Thus the throughput is roughly the same as that of infinite currency bound. In this case, tightening the refresh interval further can only hamper system performance due to unnecessary refresh. For example, for a currency bound of 100, with 15 caches, the system throughput is roughly 56, 46 and 30 for refresh interval of 50, 5 and 0.

To the other extreme, when the refresh interval is sufficiently large compared to the currency bounds, almost all read-only transactions are shifted to the master. Thus the throughput is close to that of 0 currency bound. In this case, refreshing the caches does not help. For example, for a currency bound of 5, with 10 caches, the system throughput is roughly 13 and 16 for a refresh interval of 100 and infinity.

For the best performance, the refresh interval needs to be set appropriately according to the currency bound, in order to balance the refresh overhead and local workload ratio. In Fig ? and Fig ?, we fix the currency bound to 5 and 10, respectively, and plot the system throughput for our setting of refresh intervals. While in the former case the best performance is reached with refresh interval of 0, for the latter it is with refresh interval of 5.

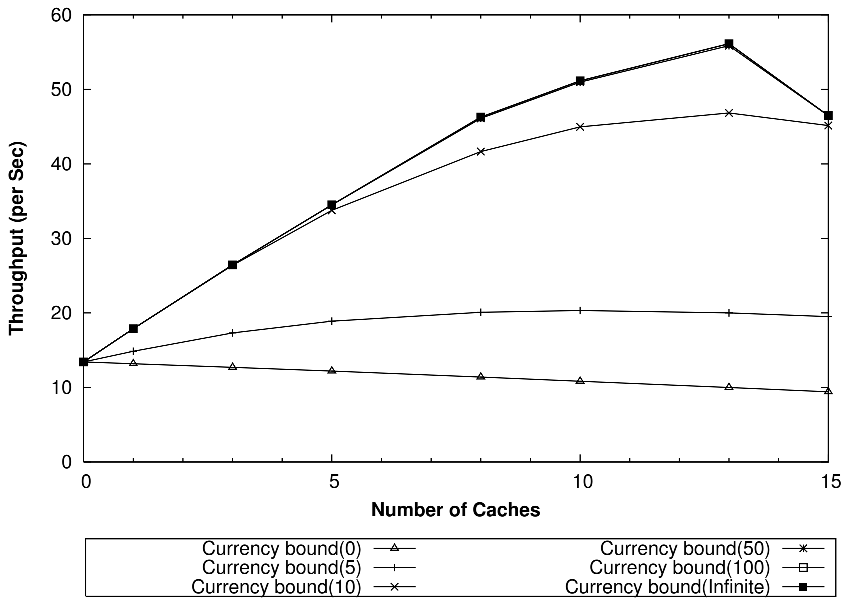


Figure 6.10 System throughput (5 refresh interval)

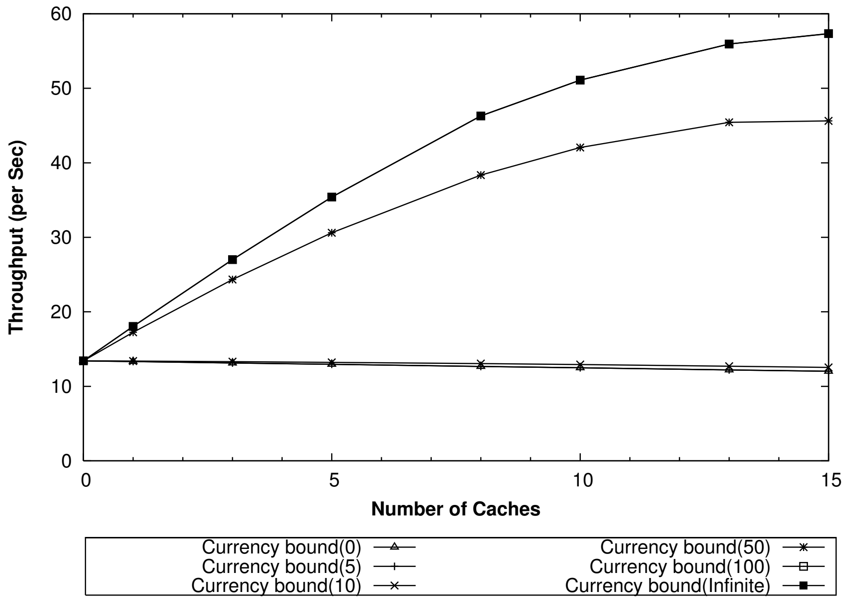


Figure 6.11 System throughput (50 refresh interval)

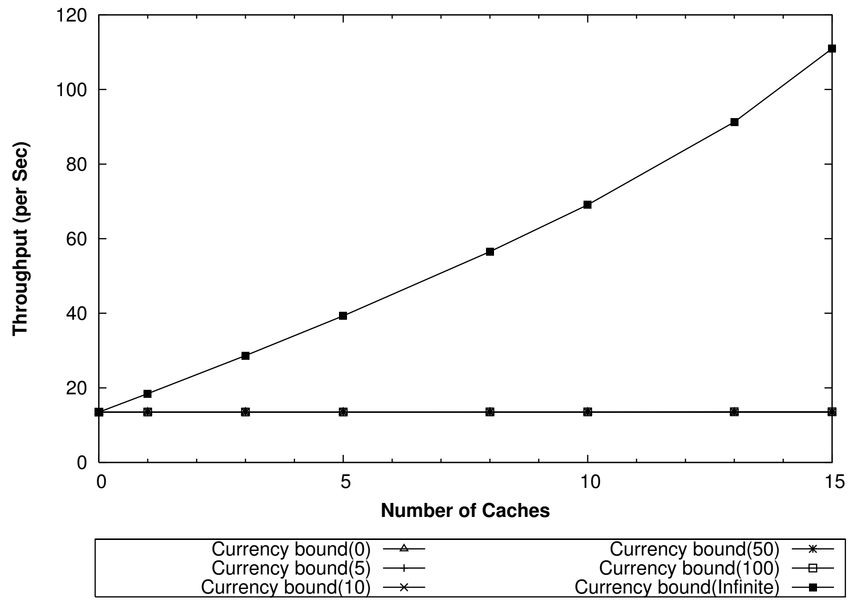


Figure 6.12 System throughput (∞ refresh interval)

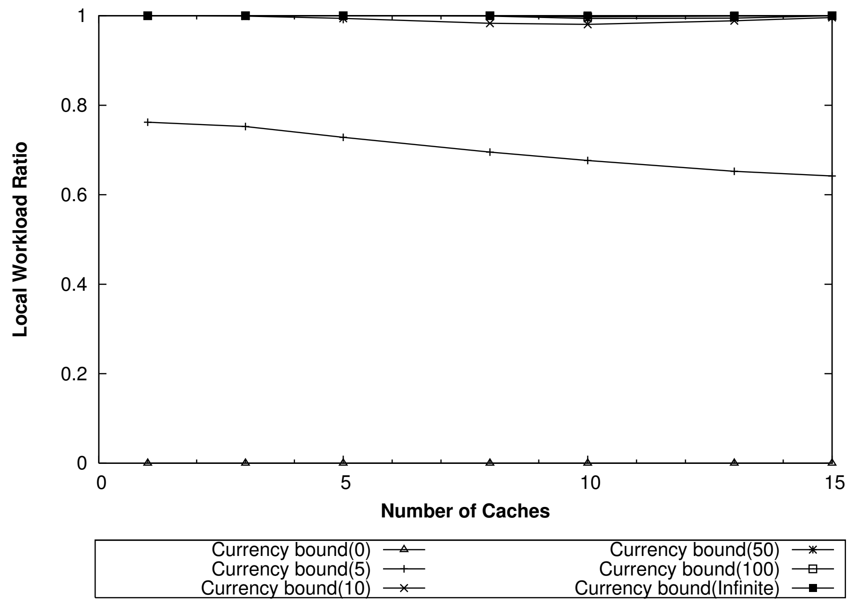


Figure 6.13 Local workload ratio for the caches (5 refresh interval)

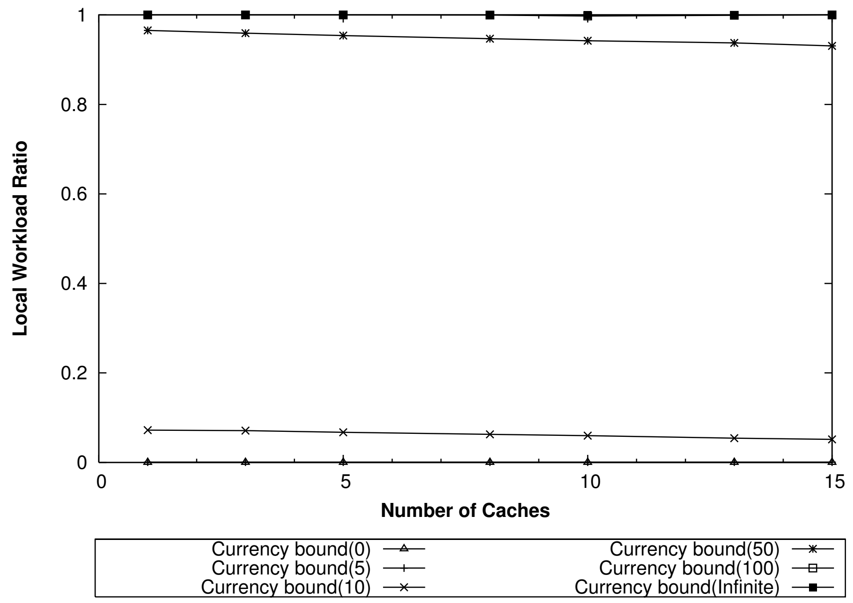


Figure 6.14 Local workload ratio for the caches (50 refresh interval)

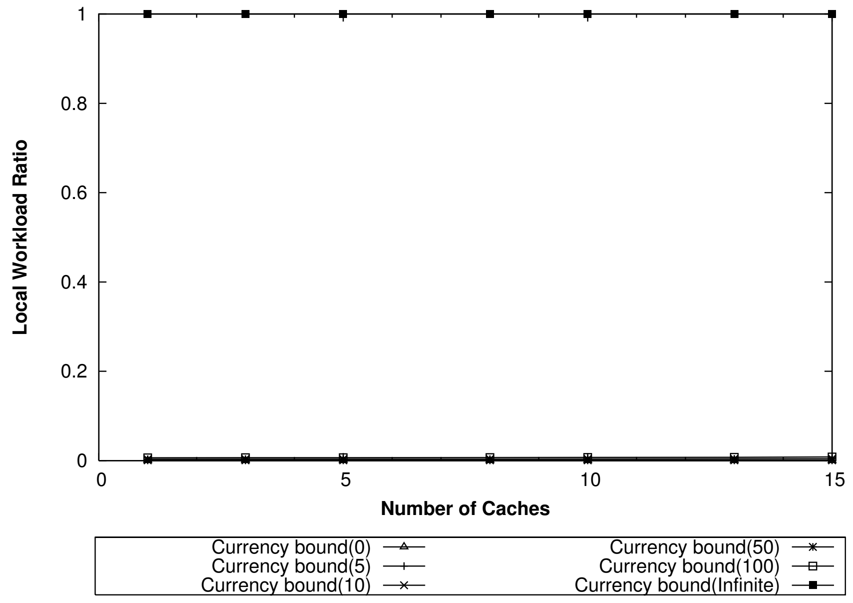


Figure 6.15 Local workload ratio for the caches (∞ refresh interval)

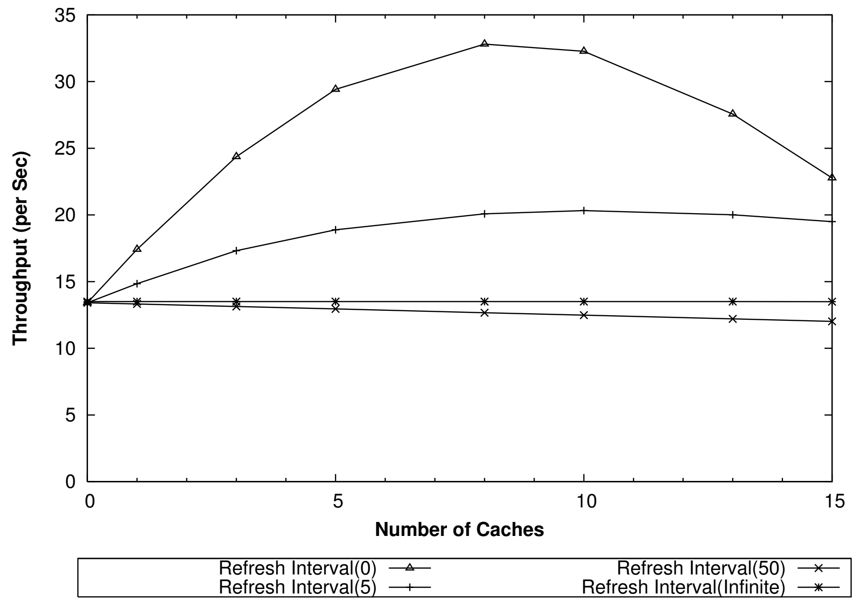


Figure 6.16 System throughput (5 currency bound)

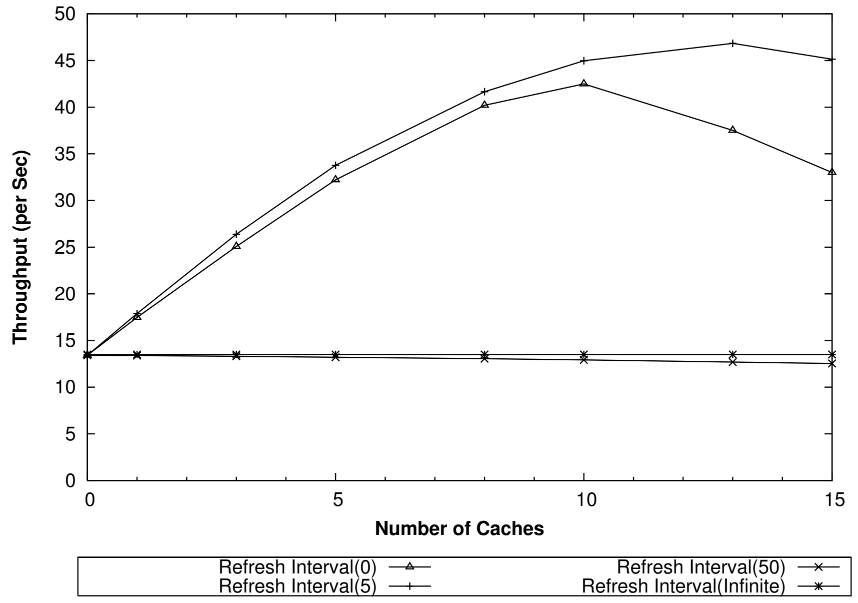


Figure 6.17 System throughput (10 currency bound)

6.5.4 Experiment 4: Impact of Push vs. Pull

In the previous experiments, we assume that the workload has the same average currency bounds for all the cache regions. In this set of experiments we study the impact of cache region granularity for workloads with skewed currency bounds. For comparison, we also use a workload with the same currency bound as a reference. For all the experiments the seed currency bound is set to 5 seconds. We first describe the experiments for the skewed setting. Then we compare that with the reference setting.

6.5.4.1 Skewed Setting

In generating the workload, we partitioned the database into 1,000 uniformly sized consistency classes. Object n belongs to consistency class with id n/10. For a skewed workload, the distribution of the average currency bounds for the regions increases quadratically. That is, region i's average currency bound is given by (i^2)*currency_bound. Regardless of the actual number of cache regions, the refresh interval of a cache region is set to the tightest average currency bound of the consistency classes that fall into this region. Thus the refresh frequency follows a Zipfian(2) distribution. Because adaptive caching was not the focus of this study, this setting assumes that the cache is tuned for the workload.

For the push model, we assign the number of cache regions to be 1, 20, 40 and 100, because it can only support view level (relatively big) cache regions. For the pull model, it is the opposite. We assign the number of cache regions to be 100, 200, 500 and 1000, because it can support fine-grained cache regions.

The skewed setting affects the system behavior under different cache region granularity mainly in two ways. Firstly, given our simulation time, according to the decaying function, out of the total 1,000 consistency classes there are 141 with non-infinity currency bound. That is, only about 14% consistency classes have non-infinity currency bounds. Secondly, as a consequence, the ratio of data at the caches that is actively maintained varies for different settings. Using some simple calculation, we compute that the numbers of active regions for the push model are 1, 3, 6, and 15, while those for the pull model are 15, 36, 71 and 141. From these numbers, the percentage of data at the caches being actively maintained is 100% for 1 with 1 cache region, 15% with 20, 40 and 100 regions, and about 14% for the rest of our settings.

Figure 6.18 shows the system throughput for the push model. Better performance is observed with more cache regions. With 10 caches, the relative improvements (compared to 1 region) are roughly 15%, 40% and 60%, for 20, 40 and 100 regions, respectively. Two factors contribute to this difference. With more cache regions, more data can be maintained exponentially less frequently and hence there is a lower distribution overhead and higher throughput for read-write transactions at the master, as shown in Figure 8.14, which in turn leads to higher throughput at the cache, as shown in Figure 8.15. Although at the same time, more regions require more distribution agents, for our settings, the difference in the number of regions is not much (1, 3, 6 and 15) compared to the difference (exponential) in refresh intervals.

For this experiment, the difference in local workload ratio at the caches is tiny, as shown in Figure 6.19. For most cases, local workload ratio is well above 0.98. Even the lowest point is still ~0.94 (1 region setting with 1 cache).

Figure 6.20 shows the throughput for the pull model with the skewed setting. For the chosen granularities, comparable throughputs are observed. Interestingly, when there are more than 10 caches, the setting with 100 regions is slightly better than that with more than 100 regions. This is because the master is burdened with the copier transactions when the number of regions increases and hence has lower throughput, as shown in Figure 8.16, which again leads to lower output to the cache (Figure 8.17). Figure 6.21 shows that in all cases almost all read-only transactions at the caches are executed locally. Thus, the difference between the throughputs is purely due to the difference in refresh cost.

Comparing the pull with the push model, there is no obvious difference with small number of caches, due to our low push maintenance overhead setting. From 10 to 15 caches, however, both with 100 cache regions, the pull model performs about 15%-60% better than the push model, where the push maintenance overhead exceeds the pull overhead.

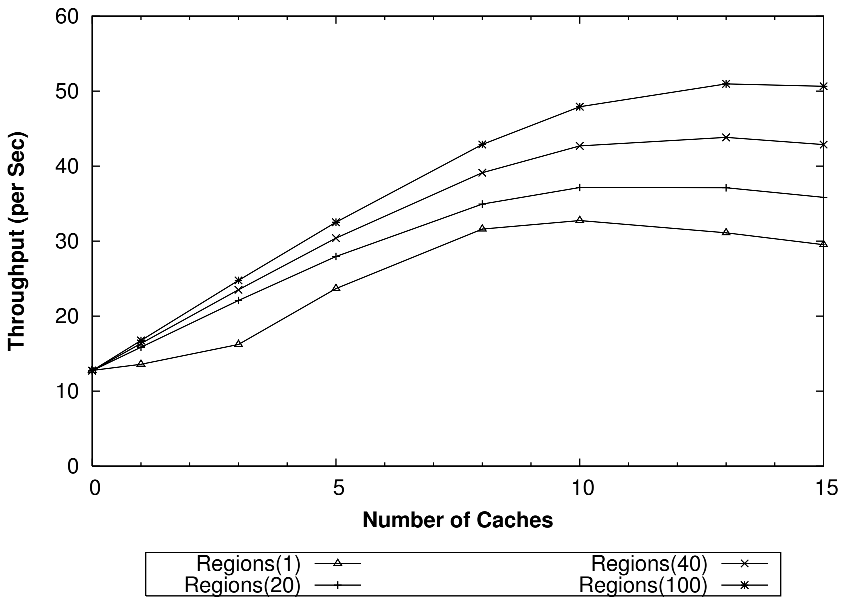


Figure 6.18 System throughput (skewed, push)

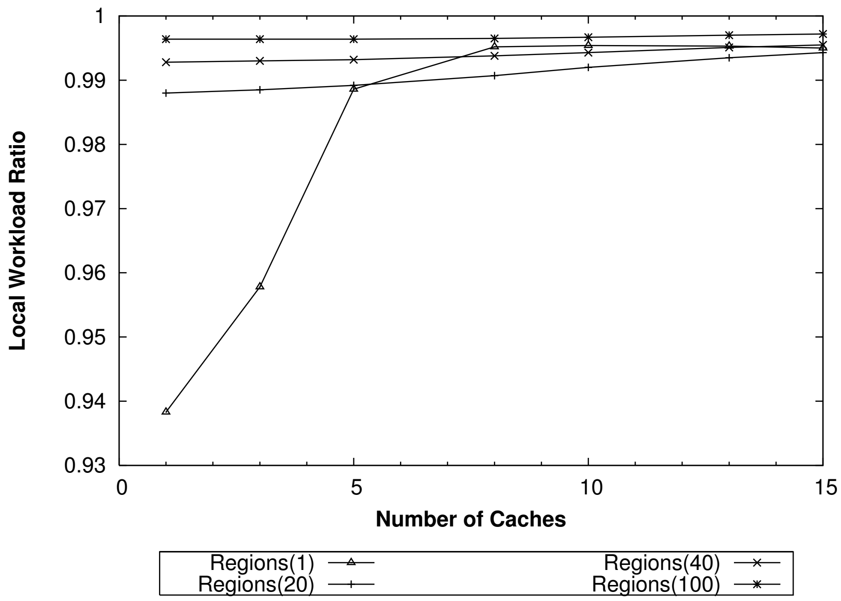


Figure 6.19 Local workload ratio at the caches (skewed, push)

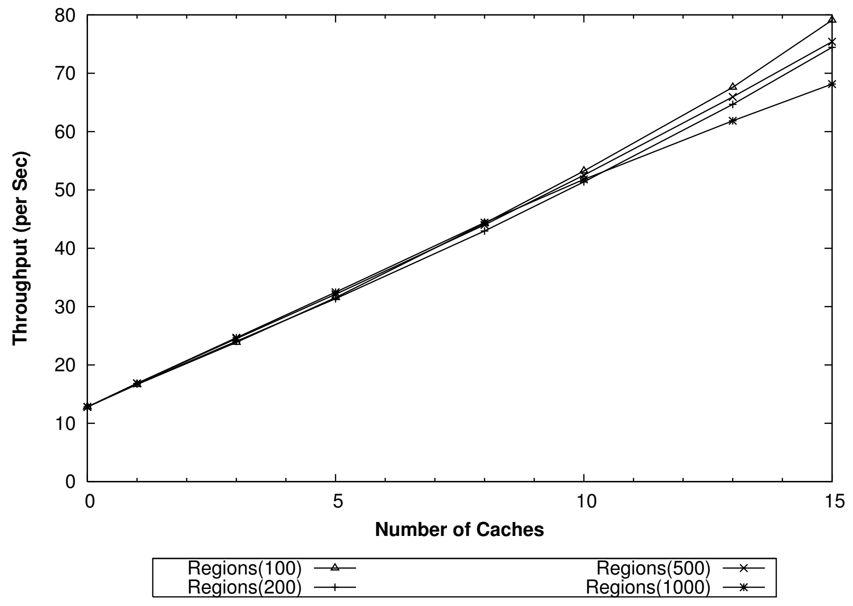


Figure 6.20 System throughput (skewed, pull)

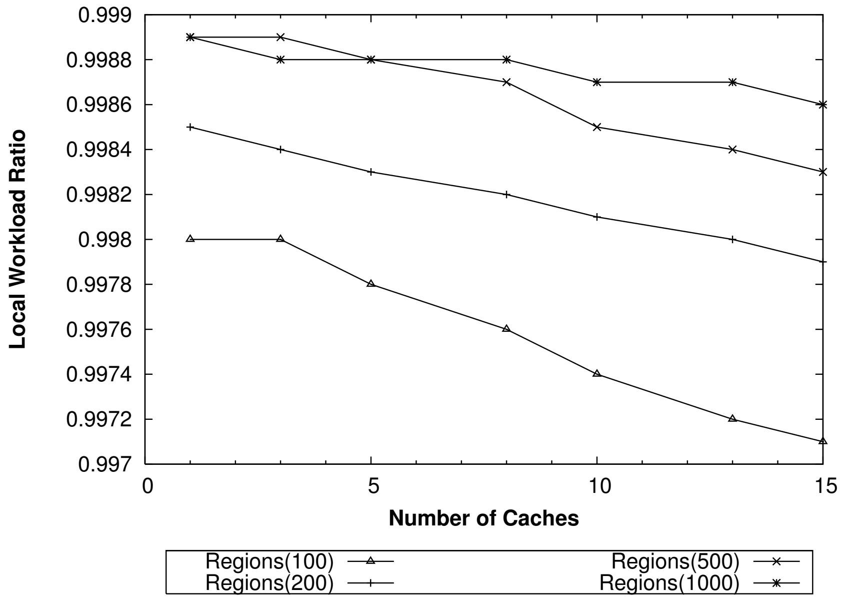


Figure 6.21 Local workload ratio for the caches (skewed, pull)

6.5.4.2 Reference Setting

For the reference workload, the refresh interval for every cache region is the same as currency_bound.

As expected, the results for the reference setting are distinct from that of the skewed setting. For the push model, as shown in Figure 6.22, the performance with 1 region improves from about 12 to 22, while that of the other settings decreases. The local workload ratio is shown in Figure 6.23. In this setting, increasing the number of regions negatively affects the performance, because it does not save any work but incurs more distribution costs. As shown in Figure 8.18 and Figure 8.19, for 20, 40 and 100 cache regions, the master throughput dominates the total throughput.

For the pull model, as shown in Figure 6.24 the performance dramatically drops below 2 when there are 3 or more caches, with all our chosen numbers of regions. This is what we expected, because with pull and 100 cache regions, it is similar to 100 more terminals that together read the whole database from the master and write to the cache. This workload makes the master the bottleneck. Because of the average high response time (above 200 seconds) of copier transactions, as shown in Figure 6.25, more than 70% of the read-only transactions are shifted back to the master, as shown in Figure 6.26, which worsens the bottleneck situation.

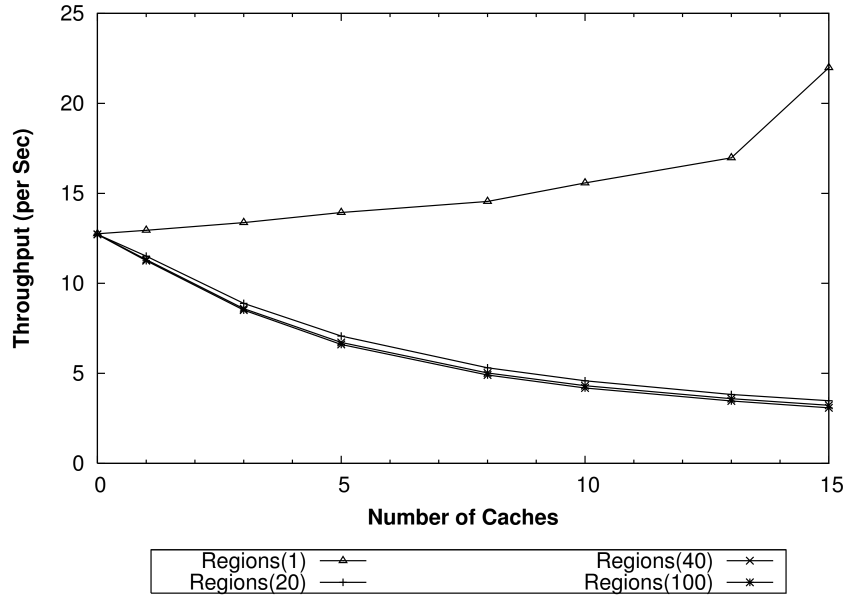


Figure 6.22 System throughput (non-skewed, push)

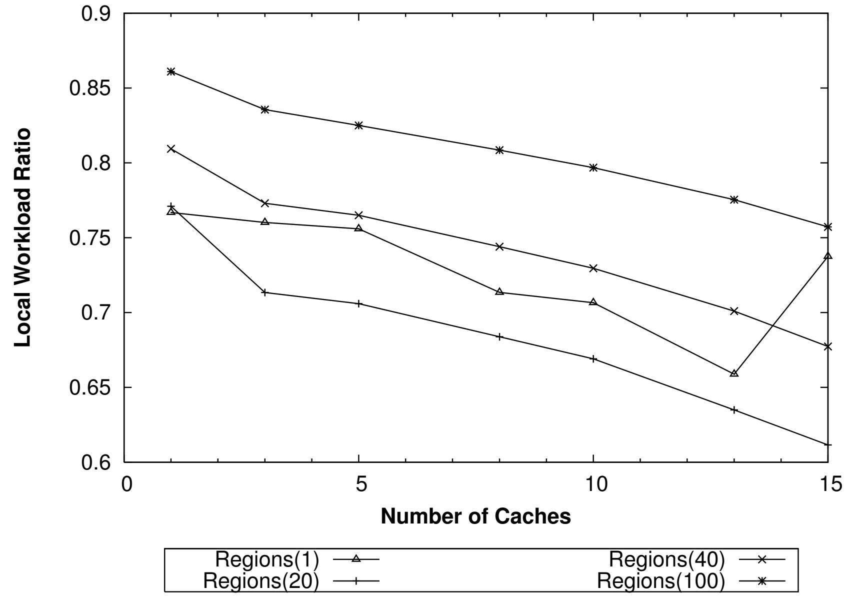


Figure 6.23 Local workload ratio for the caches (non-skewed, push)

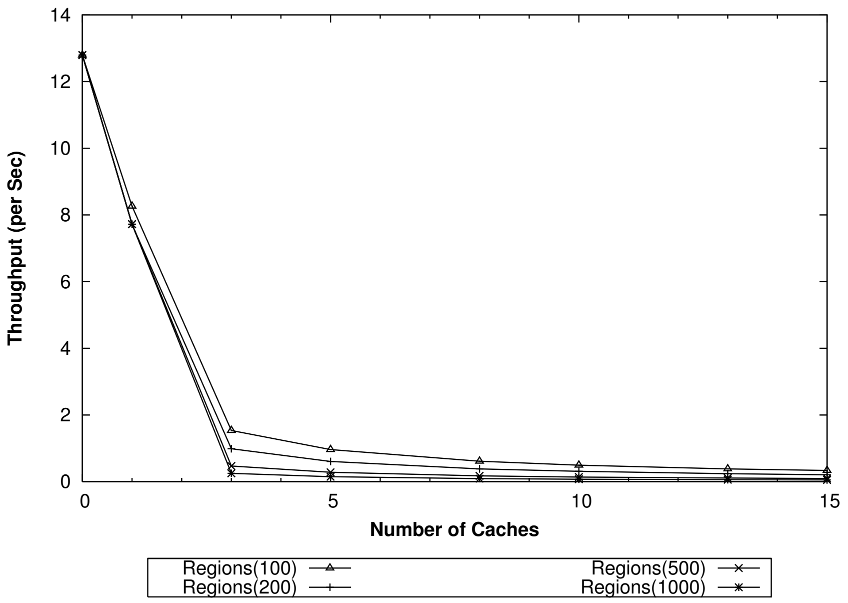


Figure 6.24 System throughput (non-skewed, pull)

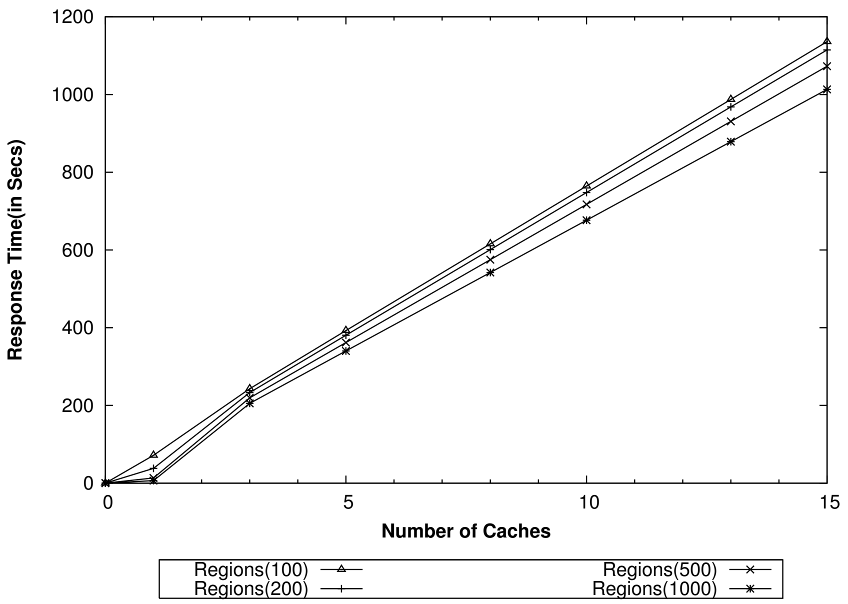


Figure 6.25 Response time for copier batch (non-skewed, pull)

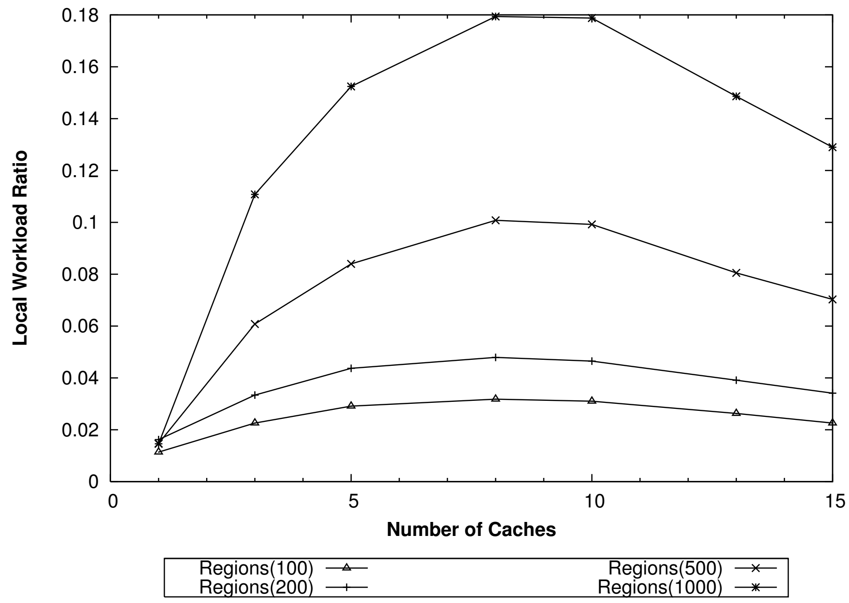


Figure 6.26 Local workload ratio for the caches (non-skewed, pull)

6.6 Conclusions

The previous chapters of this dissertation have developed a flexible data quality-aware cache management framework. We have implemented a prototype in SQL Server's code base for a simple case. In this chapter, we built a reasonable model, which allows us to explore the cache design space, and examine the system behavior in such a complex system under different assumptions.

Starting with the single-site database model from [ACL87], we extended it to cache-master configuration, and refined it to model the characteristics specific to our study. Our model includes a model of the database system and its resources, a model of the user population, a model of data transaction behavior, a cache model, a cache-master interaction model and a cache maintenance model. Except for the user model, all the others have a data quality-aware component. We built the simulation framework using the CSIM package, and included in it all of these models. We used it to study alternative assumptions about the transaction behavior and system configurations.

The first conclusion is that a detailed model is crucial in understanding the system behavior. The interaction between the user transaction data quality requirements, local data refresh policy, resource utilization and the cache maintenance overhead etc. is complex. Under different situations, different components might dominate the system performance. Only a detailed model can realistically reflect such a complex system behavior.

Specific to our setting, the first set of experiments shows that not all workload can benefit from adding more caches to the system. Even when the cache is never refreshed, we only observe reasonable performance improvement (about 6-fold with 15 caches) when the read-only percentage is above 90; with a read-only percentage of 80, for example, the performance only improves 2-fold. This is because read-write transactions cannot be processed at the caches. When the caches are refreshed continuously using the push model, even with low maintenance overhead, the performance improvement stops at 10 caches even with a read-only percentage of 90.

The second set of experiments examines the effect of different refresh intervals. Although the gap between two extreme cases with refresh interval of 0 and infinity increases with more caches, there is not much difference among the intermediate refresh intervals when the number of caches is less than 15 because of the low amortized distribution cost.

In the second set of experiments, the currency bounds are set to infinity, so the refresh interval only affects the propagation cost. Together with the low distribution cost, refresh interval does not affect system throughput that much. However, when we also provide data quality guarantees, the situation changes dramatically. The third set of experiments shows the tradeoff in refresh interval setting and its significant impact on system performance. Relaxing the refresh interval causes less refresh overhead, but more workload is shifted to the master due to poorer local data quality. Tightening the refresh interval has the opposite effects.

Thus, for best performance, the refresh interval should be carefully chosen according to queries' data quality requirements. Potential performance improvement is only possible by tightening the refresh interval if it significantly increases the local workload ratio. In our setting where distribution overhead is low, it is better to refresh the cache continuously for a workload with tight currency bounds. For a currency bound of 5, the local workload percentages are in the 90s and 70s for refresh intervals of 0 and 5, respectively, but they drop close to 0 for refresh intervals of 50 and infinity. Consequently, with 10 caches, the system throughput is 27, 19, 13 and 16 for the four refresh interval settings, respectively.

In the first three sets of experiments, we assume that the workload has the same average currency bound for the whole cache. Our last set of experiments studies the impact of cache region granularity for workloads with skewed currency bounds. For the push model, better performance is observed with more cache regions. With 10 caches, the relative improvements (compared to 1 region) are roughly 15%, 40% and 60%, for 20, 40 and 100 regions, respectively. Two factors contribute to this difference. With more cache regions, more data can be maintained exponentially less frequently and hence there is a lower distribution overhead and higher throughput for read-write transactions at the master, which in turn leads to higher throughput at the cache.

For the pull model, for the chosen granularities, comparable throughputs are observed. Peak performance is obtained with 100 cache regions when there are more than 10 caches. This is because the master is burdened with the copier transactions when the number of regions increases and hence has lower throughput, which again leads to lower output to the caches.

Comparing the pull with the push model, for a read-dominated workload with similar currency bounds, the push model with 1 cache region is more efficient. The pull model with fine granularity of cache regions excels for a skewed workload where only a small part of the cache is accessed with high data quality requirements. There is no obvious difference with a small number of caches, due to our low push maintenance overhead setting. From 10 to 15 caches, however, both with 100 cache regions, the pull model performs about 15%-60% better than the push model, where the push maintenance overhead exceeds the pull overhead.

A more general conclusion is that synchronization delay may lead to a drop in local workload ratio; hence special care has to be given to the copier and refresh transactions. It is not a good practice to combine many small refresh transactions into one, because of the longer lock waiting time and the higher chance of lock failure and hence surged response time. Further, it is necessary to give refresh transactions higher priority over user transactions. Otherwise, even for a workload with a low write probability, the response time of a refresh transaction may surge due to resource contention.

Chapter 7

Related Work

We classify work related to ours into two main categories: work on relaxing data quality and work on caching. In comparison with them, our approach is unique in the following ways: 1) we allow individual queries to specify fine-grained consistency and currency requirements; 2) we allow flexible local data quality control in terms of granularity and cache properties; and 3) we provide transactional C&C guarantees for individual queries.

7.1 Relaxing Data Quality

Tradeoffs between data freshness and availability, concurrency and maintenance overhead have been explored in several areas of database systems, such as replica management, distributed databases, warehousing and web caching. Yet no work we know of allows queries to specify fine-grained C&C constraints, provides well-defined semantics for such constraints, and produces query plans guaranteeing that query results meet the constraints.

7.1.1 Replica management

In a typical replica system setting, updates are centralized on a back-end server, while read workloads are offloaded to local replicas. Keeping all the copies up to date at all times is neither practical nor necessary. Can one lower the maintenance overhead at the cost of lower data quality? Different studies have tackled different aspects of this problem.

Quasi-copies [ABGMA88] allow an administrator to specify the maximum divergence of cached objects, and maintain them accordingly. A later paper [GN95] formalizes these concepts and models the system using a queuing network. The work on “Good Enough” Views [SK97] extends these ideas to approximate view maintenance; Globe [KKST98] to wide-area distributed systems; [LC02] to mobile computing with distributed data sources. Identity connection [WQ87] suggests a relationship to model the connection between a master and its copies. Researchers at Bellcore [SR90] proposed taxonomy for interdependent data management.

The approach taken in these papers is fundamentally different from ours: their approach is *maintenance centric* while ours is *query centric*. They propose different approximate replica maintenance policies, each guaranteeing certain C&C properties on the replicas. In contrast, given a query with C&C requirements, our work focuses on extending the optimizer to generate a plan according to the known C&C properties of the replicas. Thus, C&C requirements are enforced by the cache DBMS.

TRAPP [OW00] stores intervals instead of exact values in the database, combining local bounds and remote data to deliver a bounded answer that satisfies the precision requirement.

Divergence caching [HSW94], Moving Objects Databases [WXCJ98] and work at Stanford [OLW01] deal with the problem of setting optimal bounds for approximate values given queries with precision bound and an update stream. These earlier query-centric proposals allow a query to specify divergence bounds and guarantee that the bounds are met. However, they have several limitations. First, they do not guarantee any consistency. Second, they do not consider using derived data, e.g., materialized views, to answer queries. Third, field-value level currency control limits the scalability of those systems. Fourth, the decision to use local data is not cost based, i.e., local data is always used if it satisfies the currency constraints.

7.1.2 Distributed databases

In this area there are many papers focused on improving availability and autonomy by allowing local data to diverge from the master copy. They differ from each other in divergence metrics, the concrete update protocols and corresponding guaranteed divergence bound. Read-only Transactions [GMW82], the Demarcation Protocol [BGM92], TACC [YV00a, YV00b], and ASPECT [Len96] fall into this category. These researches are all maintenance centric. None of them supports queries with data quality requirements.

Epsilon-serializability [PL91] allows queries to specify inconsistency bounds. However, they focus on a different problem, hence utilize different techniques: how to achieve higher degree of concurrency by allowing queries to see database states with bounded inconsistency introduced by concurrent update transactions.

7.1.3 Warehousing and web views

WebViews [LR03] suggests algorithms for the on-line view selection problem considering a new constraint — the required average freshness of the cached query results. Obsolescent Materialized Views [Gal99] determines whether to use local or remote data by integrating the divergence of local data into the cost model of the optimizer. A later paper [BR02] tackles a similar problem for single object accesses. In all these approaches, the models of freshness are coarse-grained and its use is purely heuristic, providing no guarantees on delivered data currency and consistency.

The work on distributed materialized views in [SF90] allows queries to specify currency bounds, and they also support local materialized views. However, it focuses on determining the optimal refresh sources and timing for multiple views defined on the same base data. It does not consider consistency constraints, assuming a query is always answered from a single view. Furthermore, it is not clear how it keeps track of the currency information of local views, or how and when it checks the currency constraints.

FAS [RBSS02] explores some preliminary query-centric ideas by allowing queries to specify currency requirements. Working as middleware on top of a cluster of multi-versioned replicated databases, FAS provides two major functionalities: (1) routing a query to the right database according to its currency requirement, and (2) deciding when and which replica database to refresh based on the workload with currency requirements. Compared to our work, FAS has three major limitations. First, it does not allow queries to specify relaxed consistency requirements, i.e., a query result always has to be consistent. Second, it only supports database level currency control. This limits replica maintenance flexibility, possibly resulting in higher overhead. Last but not least, enforcing currency requirements at the middleware level instead of inside the DBMS, FAS cannot provide transactional currency guarantees on query results.

7.2 Semantic Caching

Data-shipping at page level have been studied primarily in the context of object-oriented database systems, detailed discussion regarding this body of research can be found in [Fra96]. The work in [DFJ+96] proposes a semantic caching model, in which the cache is organized into semantic regions. Usage information and replacement policies are maintained for semantic regions. In general, a semantic caching system needs to address the following three fundamental questions: 1) Replacement / admission policies? 2) How to map a query to the cached data? 3) How to maintain the cache in the presence of updates? We first summarize related work from each of the above perspectives respectively. Then we address the work most closely related to ours in the middle tier caching context. To the best of our knowledge, there is no previous work on semantic caching that is tailored for workload with relaxed C&C constraints.

Caching has been used in many areas. Regarding what to cache, while some works [DFJ+96, APTP03] support arbitrary query results, others are tailored for certain simple types of queries [KB96, BPK02, LN01], or even just base tables [AJL+02, CLL+01, LKM+02]. In the database caching context, good surveys can be found in [DDT+01, Moh01].

7.2.1 Replacement and Admission Policy

From a cache admission and replacement policy perspective, quite a few policies are proposed in the literature, each of which tries to maximize a certain optimization goal under a given space budget. Least Recently Used (LRU) is probably the most widely used cache replacement policy. As an extension, LRU-K [OOW93] intends to discriminate between objects that have frequent references and objects that are rarely accessed by looking at a longer history. However, LRU-based schemes are not suited for a semantic cache because they assume that all objects are of the same size, same miss penalty and same replacement cost. As a remedy, size-adjusted LRU (SLRU) [AWY99] is designed to generalize LRU to objects of varying sizes and varying replacement costs.

The work in [DFJ+96] uses semantic distance for the replacement policy. However, this metric is only proper when the semantic regions have well defined geometric meaning.

In [SJGP90], a cache management system is described for Postgres. The proposed admission and eviction policies take into account several properties of the cached views: size, cost to materialize, cost to access it from the cache, update cost, access frequency, and update frequency. These parameters are combined in a formula that estimates the benefit per unit size of caching a view. A similar cache management mechanism is used by [KB96], but in a different context, namely, semantic caching in a client-server architecture.

DynaMat [KR99], a dynamic view selection and maintenance system is proposed in the OLAP context. Several "goodness" functions are evaluated. Two constraints are considered: time bound for update and space bound for admission and replacement.

WATCHMAN [SSV96] is a warehouse caching system. The proposed profit function combines average rate of reference, size, and cost of execution of a query. Their admission and eviction policies are similar to the Postgres policies. Similar to LRU-K, they also retain, for some time, information about results that have been evicted from the cache. A follow-up work [SSV99] includes partial reuse and considers updates.

The cache investment idea is introduced in [KFDA00] for a distributed database system. Different from our work, they consider caching base tables and indices, hence different techniques are used to gather statistics and to generate candidates.

In [SS02], in a multi-version caching system, a future access frequency is estimated from the frequency and recency of past references. In the cost/benefit function, they consider cost of aborts due to a miss of an object version that is not re-fetchable.

7.2.2 Matching Cached Results

In the semantic caching scenario, the proposed matching algorithms for cached results are mostly tailored to certain knowledge of the cached data. In [LC99] knowledge-based rules are used as an addition to the match algorithms. Matching algorithms for conjunctive queries are proposed in [CB00]. The form-based caching scheme [LN01] describes containments test for queries from a simple top-n selection on a single table view with a keyword predicate.

The improved version of WATCHMAN [SSV99] recognizes two concepts of derivability, namely, 1) a data cube is derivable from a compatible but less aggregated data cube and b) a slice query is derivable from its underlying data cube query. For all other queries, exact match is required. The system maintains query derivability information in a directed graph. Similarly, DynaMat [KR99] also keeps the data cube lattice for derivability information. In addition, they use a Directory Index for each node to further prune the search space.

In a chunk-based caching scheme for OLAP systems [DRSN98], a level defined by a group by is divided into uniform chunks, which are the unit of caching. A query can be answered only from the same level of chunks. A follow-up work [DN00] relaxes this constraint by allowing a query to use chunks from lower levels of aggregation. They associate a Virtual Count with each chunk that records the derivability information of the chunk. The algorithm also maintains cost-based information that can be used to figure out the best possible option for computing a query result from the cache.

7.2.3 Cache Maintenance

From a cache maintenance policy perspective, existing works on update propagation and cache invalidation can be in turn classified into two categories: shared-storage and shared-nothing based. As an example of the former, [APTP00] and [ATPP02] conceptually decompose the query results and store the tuples in the base tables they are from. The idea is that to maintain base tables is cheaper than to maintain a set of views. However, with this approach, queries sent to the back-end have to be rewritten and the eviction becomes expensive. Furthermore, we lose the flexibility to maintain the cache according to different C&C requirements.

Algorithms in the second category are complementary to our approach. We focus on how to reduce maintenance frequency with the knowledge of relaxed C&C constraints, while they focus on how to propagate updates or calculate invalidation efficiently. [LR01] proposed view updates scheduling algorithms for the back-end server to maximize QoD (Quality of Data) based on the cost of updating and popularity of views. [CDI99] used the DUP algorithm to maintain data dependence information between cached objects and the underlying data, and built a hash table index to efficiently invalidate or update highly obsolete cached objects. [CLL+01] shows an example of using popular components in the industry to support invalidation of front-end cached web pages. [CAL+02] suggests using polling queries to reduce server invalidation workload at the cost of over-invalidation.

7.2.4 Middle-tier Database Caching

In the context of middle-tier caching, the closest work to ours are DBCache [ABK+03] and Constraint-based Database Caching (CBDC) [HB04]. Similarly to us, they consider full-fledged DBMS caching; and they define a cache with a set of constraints. However, there are two fundamental differences. First, they don’t consider relaxed data quality requirements, nor do they provide currency guarantees from the DBMS. Our work is more general in the sense that the cache-key and RCC constraints (an extension to cache groups in [Tea02]) they support can be seen as a subset of ours. Second, in DBCache, local data availability checking is done outside of the optimizer, while in our case, local data checking is integrated into query optimization, which not only allows finer granularity checking, but also gives the optimizer the freedom to choose the best plan based on cost.

Chapter 8

Conclusion and Future Directions

This work was motivated by the lack of a rigorous foundation for the widespread practice of applications using replicated and cached data. This dissertation remedied the situation by extending DBMS support for weak consistency, realizing our vision: applications are allowed to explicitly express relaxed currency and consistency (C&C) requirements in SQL; an administrator can explicitly express the desired local data C&C level in the cache schema; and query processing provides transactional guarantees for the C&C requirements of a query.

In Chapter 2, we showed how C&C constraints can be expressed succinctly in SQL through a new currency clause. We developed a formal model that strictly defines the semantics of general C&C constraints, providing correctness standards for the use of replicated and cached data.

In Chapter 3, we presented a fine-grained C&C-aware database caching model, enabling a flexible cache administration in terms of granularity and desired local data quality. We formally defined four fundamental cache properties: presence, consistency, completeness, and currency. We proposed a cache model in which administrator can specify a cache schema by defining a set of local views, together with cache constraints that define what properties the cache must guarantee.

In Chapter 4, we demonstrated for a simplified case, where all the rows in a view are consistent, how support for C&C constraints can be implemented in MTCache, our prototype mid-tier database cache built on the Microsoft SQL Server codebase. We not only fully integrated C&C checking into query optimization and execution, enabling transactional guarantees, but also provide cost estimation, giving the optimizer the freedom to choose a best plan based on cost.

In Chapter 5, we extended the framework developed in Chapter 4 to enforce C&C checking for a finer-grained cache model, where cache properties are defined at the unit of partition of a view.

Chapter 2 - Chapter 5 presented a comprehensive solution for a flexible data quality aware cache management. In Chapter 6, we built a cache-master simulator based on a closed queuing model, and conducted a series of experiments, systematically evaluating and comparing the effectiveness of different design choices, offering insights into performance tradeoffs.

To conclude, this dissertation built a solid foundation for supporting weak consistency in a database caching system, and we have demonstrated the feasibility of the proposed solution by implementing a prototype in the Microsoft SQL Server codebase. We envision three directions of future research:

To improve the current prototype. Firstly, now we only process read-only transactions at the cache. One possible future work is to handle read-write transactions at the cache. A possible extension is to process the read part of a read-write transaction at the cache, and send the write part to the master. However, it is not unusual for an application to require seeing its own changes, which imposes time-line constraints on the transaction. How do we efficiently support such constraints? Secondly, in our current cache model, we only support groups defined by equality conditions. For efficient cache management, it would be useful to include other predicates, e.g., range predicates.

Adaptive data quality-aware caching policies. This dissertation developed mechanisms for a flexible fine-grained cache model, but not the policies. How do we adaptively decide the contents of the control tables? What refresh intervals to set for each cache region? Allowing queries to specify relaxed C&C constraints fundamentally changes the traditional semantic caching problem in two ways. First, hit ratio is no longer a good indicator of cache usage. The fact that a query can be semantically answered from the cached data does not necessarily mean that the cache can be used, because local data might not be good enough for the query’s C&C constraints. Second, update workload on the backend database no longer directly decides the maintenance overhead of the cache. For instance, even if the backend database is updated frequently, we do not have to propagate updates to the cache as often if the query workload allows extremely relaxed C&C constraints.

Automatic/aided cache schema design and tuning. Given a query workload with C&C constraints and an update workload at the master, how to generate a good cache schema? How to adjust a cache schema when the workload changes? Currently the cache admin is burdened with this task. To devise algorithms to aid or even automate this process remains challenging.

Appendix

Supporting Graphs

Experiment 1:

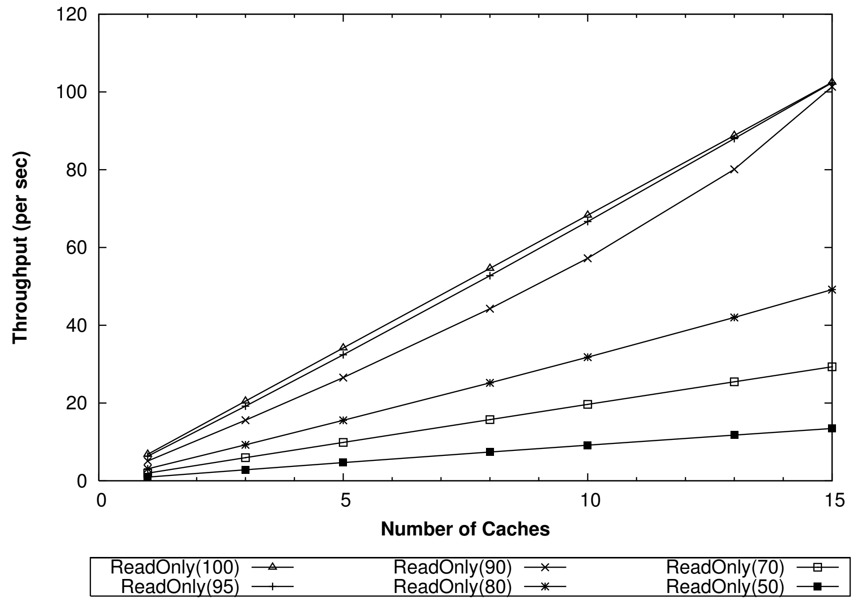


Figure 8.1 Throughput for all the caches
(∞ currency bound, ∞ refresh interval)

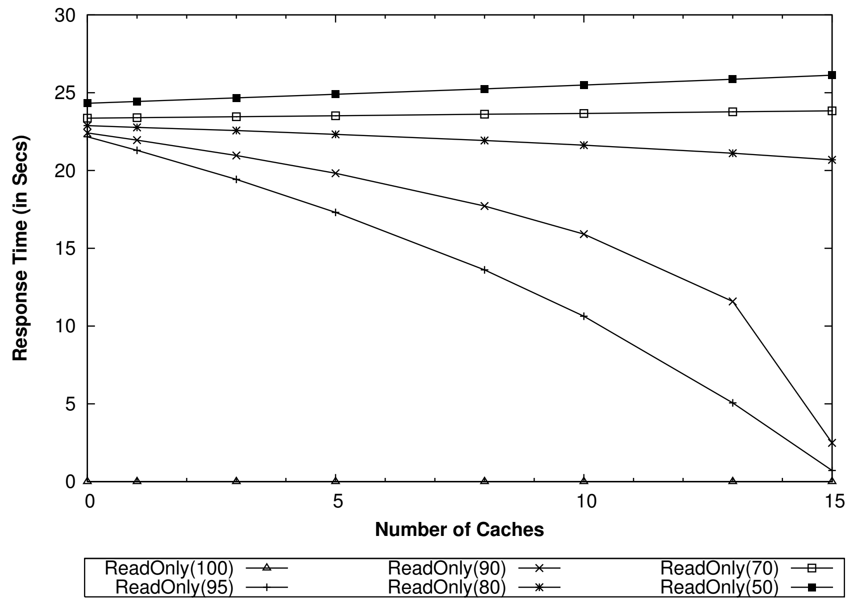


Figure 8.2 Response time for read-write transactions at the Master (∞ currency bound, ∞ refresh interval)

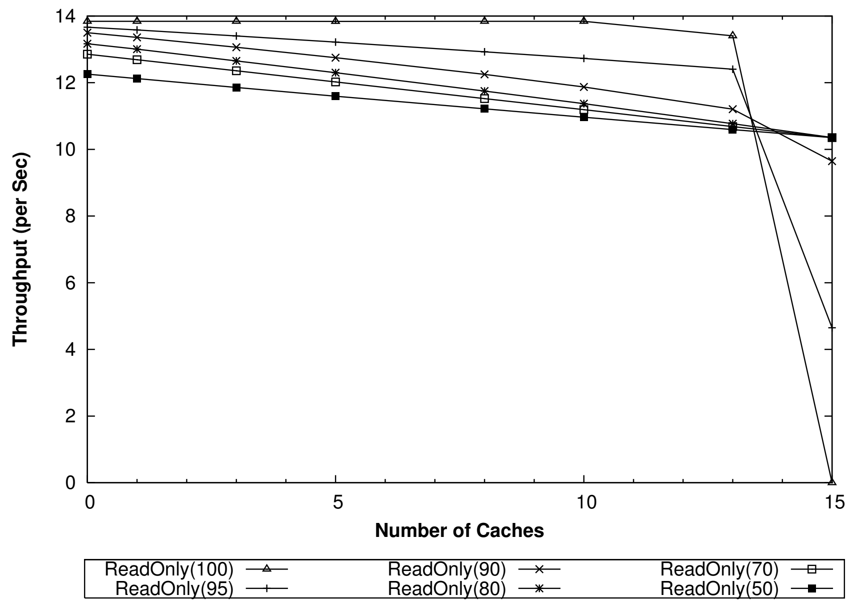


Figure 8.3 Master throughput (∞ currency bound, ∞ refresh interval)

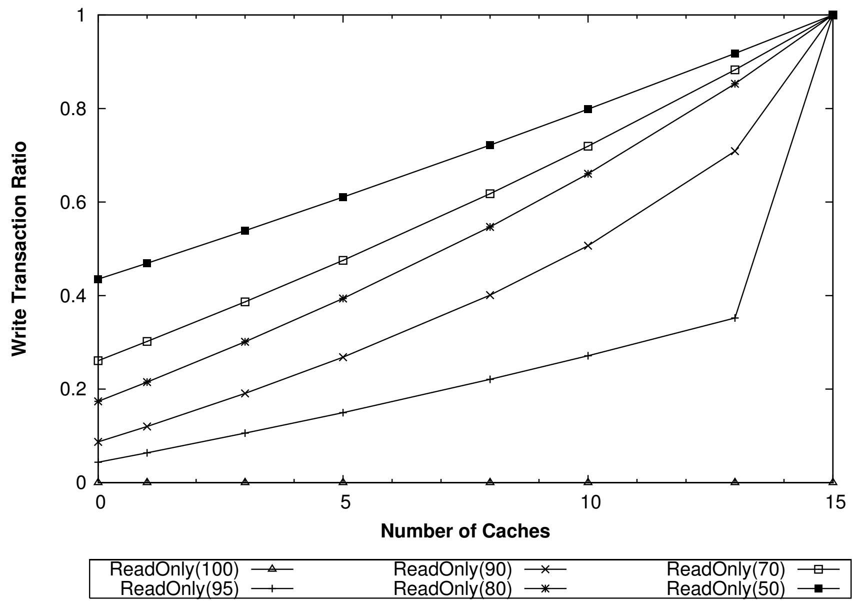


Figure 8.4 Read-write transaction ratio at the master (∞ currency bound, ∞ refresh interval)

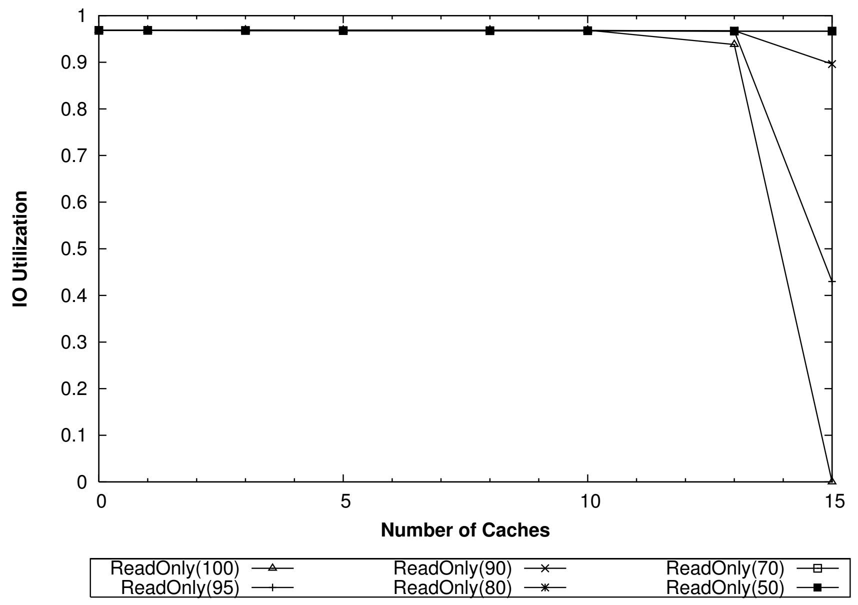


Figure 8.5 Disk utilization for the master (∞ currency bound, ∞ refresh interval)

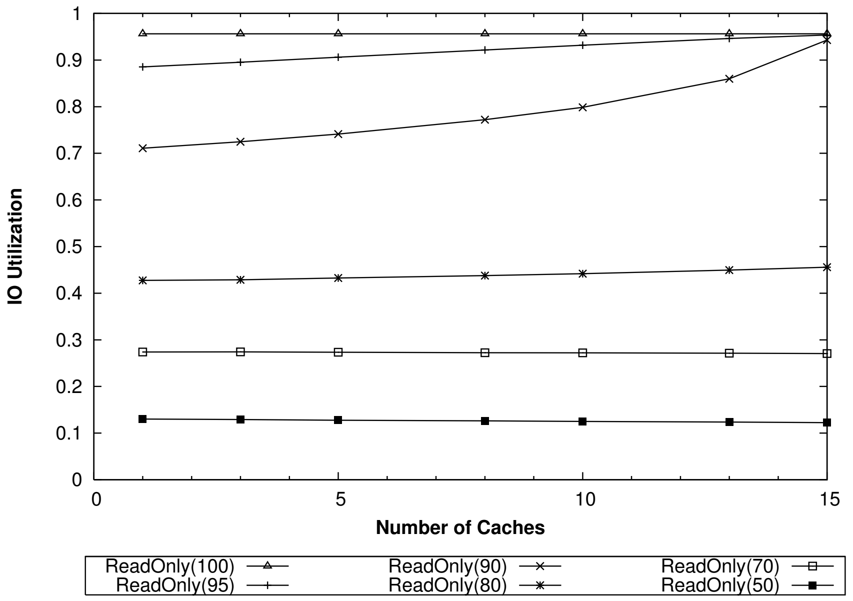


Figure 8.6 Disk utilization for the caches (∞ currency bound, ∞ refresh interval)

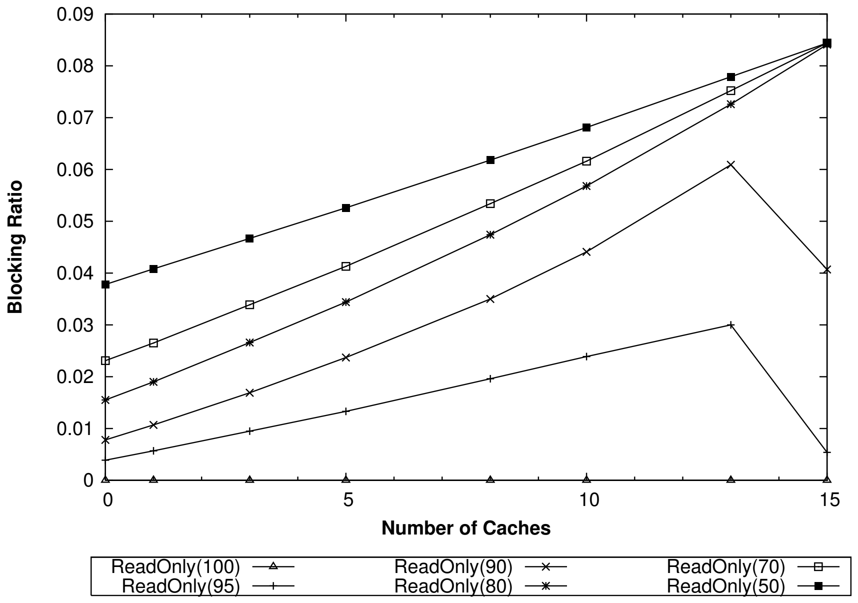


Figure 8.7 Blocking ratio for the master (∞ currency bound, ∞ refresh interval)

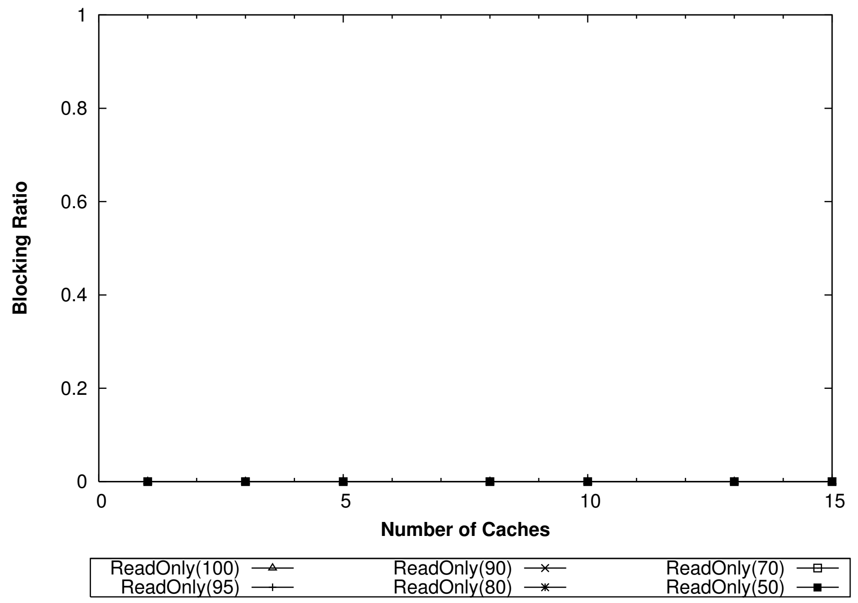


Figure 8.8 Blocking ratio for the caches (∞ currency bound, ∞ refresh interval)

Experiment 2:

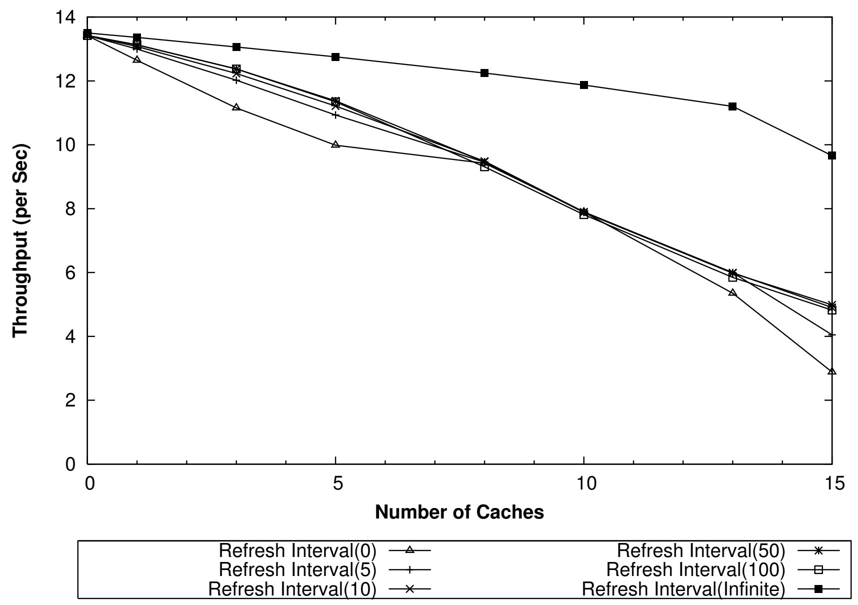


Figure 8.9 Master throughput (∞ currency bound)

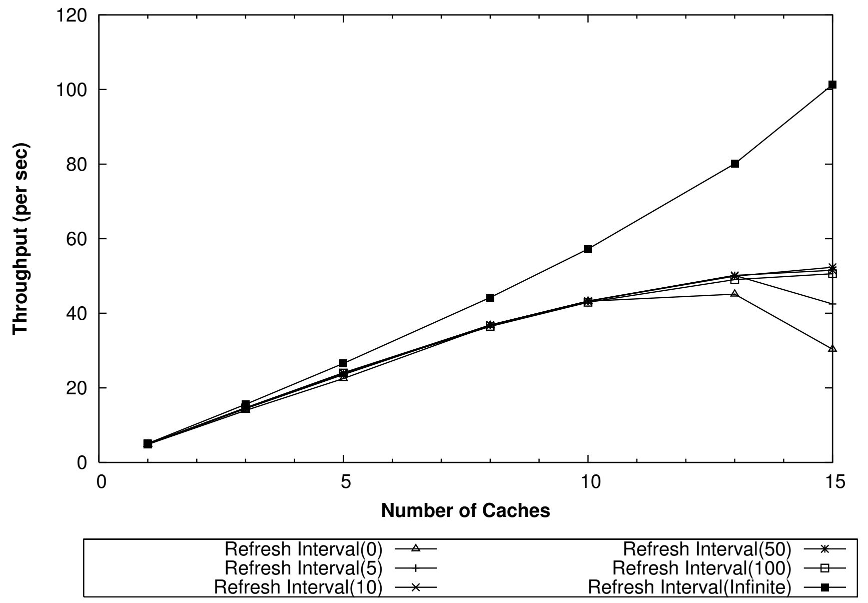


Figure 8.10 Throughput for all caches (∞ currency bound)

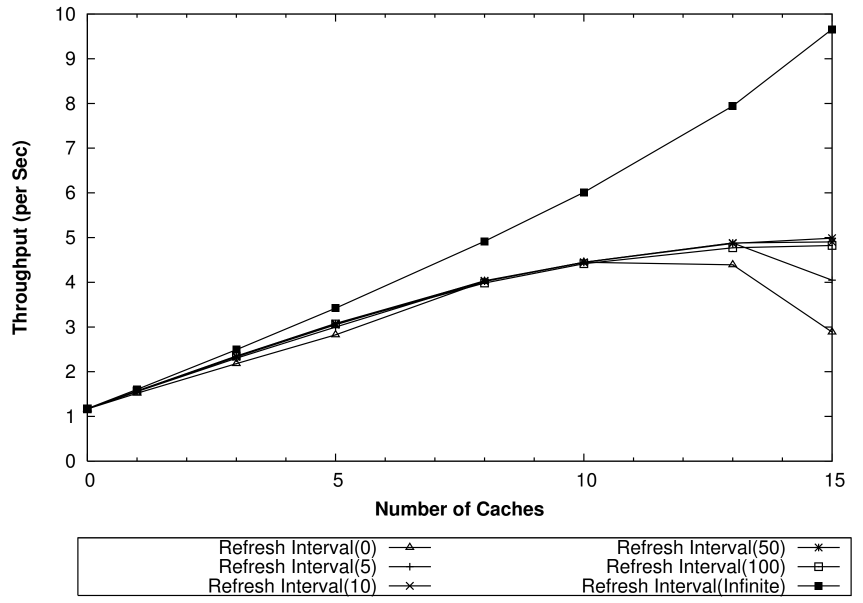


Figure 8.11 Throughput for read-write transactions at the Master (∞ currency bound)

Experiment 3:

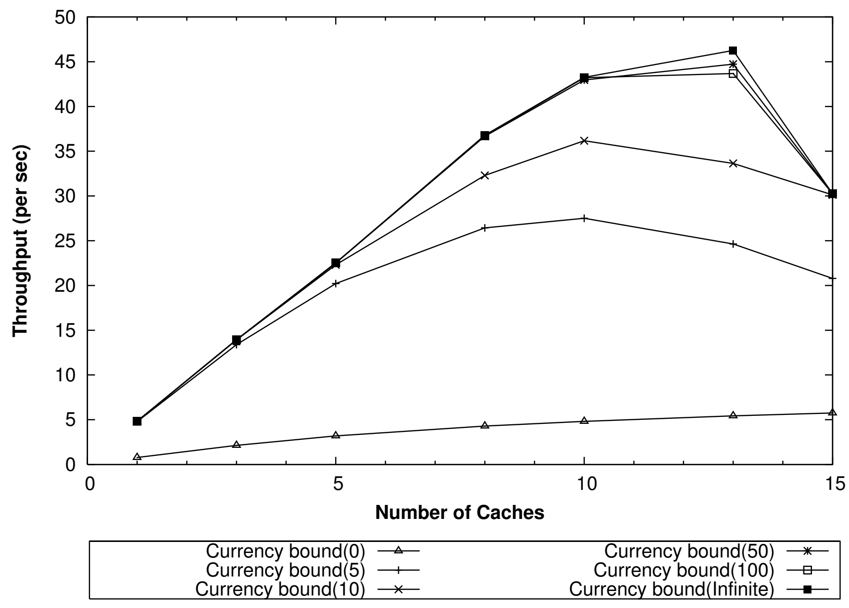


Figure 8.12 Throughput for all caches (0 refresh interval)

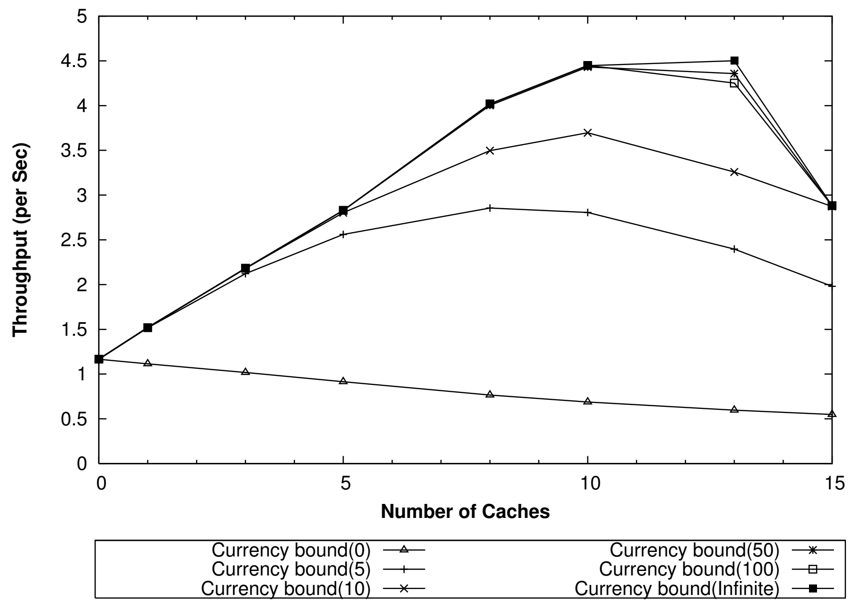


Figure 8.13 Throughput for read-write transactions at the master (0 refresh interval)

Experiment 4:

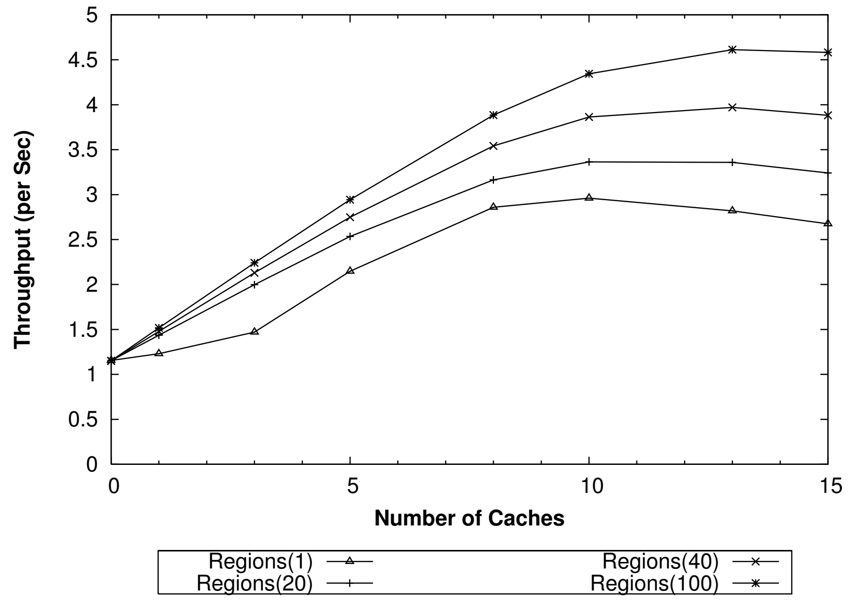


Figure 8.14 Throughput for read-write transactions at the master (skewed, push)

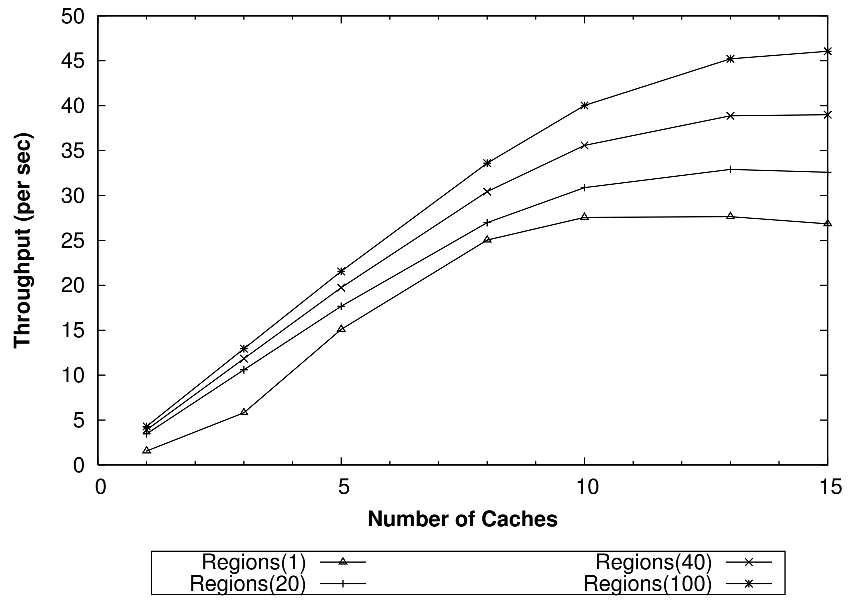


Figure 8.15 Throughput for all caches (skewed, push)

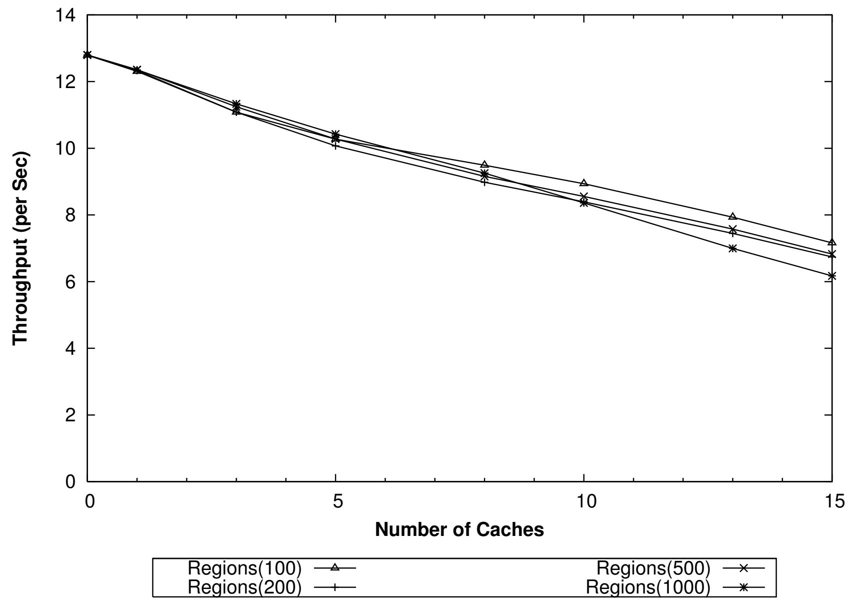


Figure 8.16 Master throughput (skewed, pull)

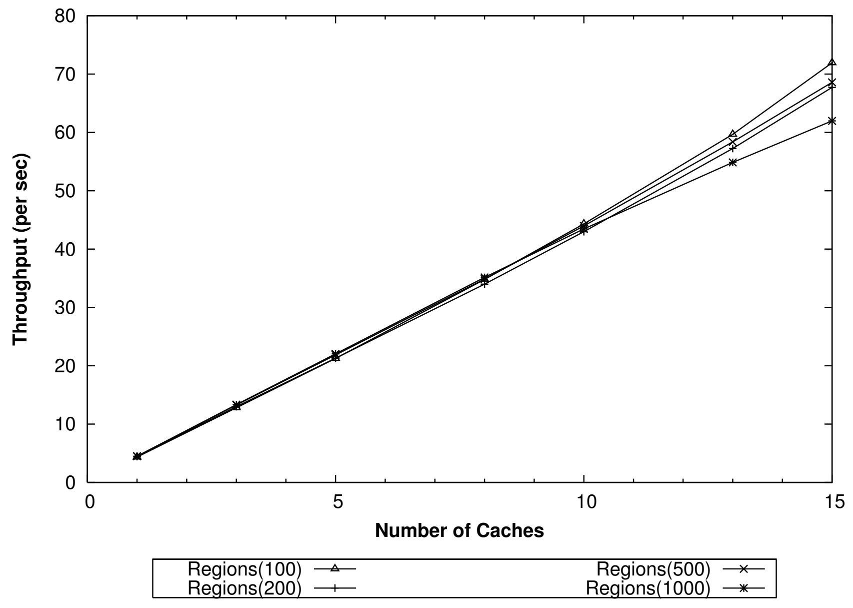


Figure 8.17 Throughput for all caches (skewed, pull)

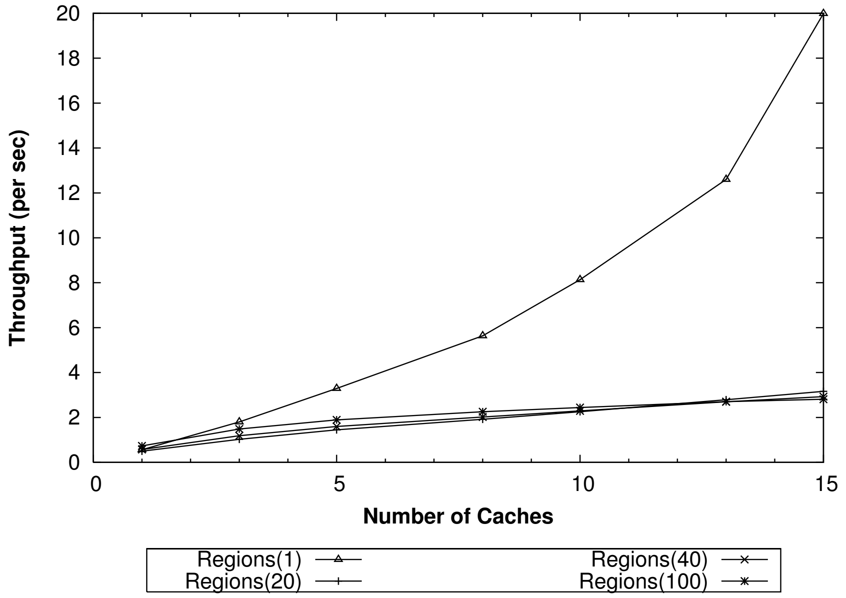


Figure 8.18 Master throughput (non-skewed, push)

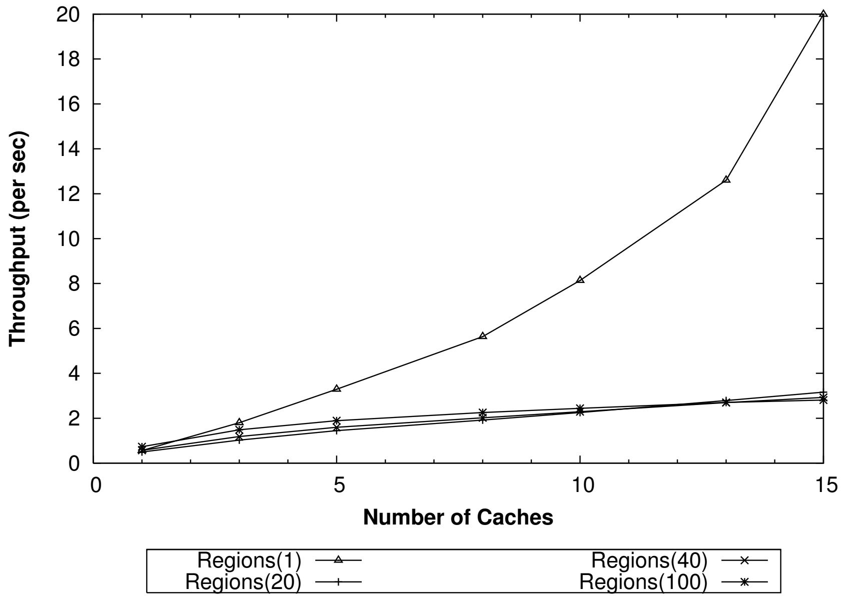


Figure 8.19 Throughput for all caches (non-skewed, push)

Bibliography

[ABGMA88]Rafael Alonso, Daniel Barbará, Hector Garcia-Molina, and Soraya Abad. Quasi-copies: Efficient data sharing for information retrieval systems. In *Proceedings of the International Conference on Extending Database Technology(EDBT)*, Venice, Italy, March 14-18, 1988, volume 303 of Lecture Notes in Computer Science, pages 443–468. Springer, 1988.

[ABK+03] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C.Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 718–729, Berlin, Germany, September 2003.

[ACL87] Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems (TODS)*, 12(4):609–654, 1987.

[AJL+02] Jesse Anton, Lawrence Jacobs, Xiang Liu, Jordan Parker, Zheng Zeng, and Tie Zhong. Web caching for database applications with oracle web cache. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 594–599, New York, NY, USA, June 2002. ACM Press.

[APTP03] Khalil Amiri, Sanghyun Park, Renu Tewari, and Sriram Padmanabhan. Dbproxy: A dynamic data cache for web applications. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 821–831, 2003.

[ATPP02] Khalil Amiri, Renu Tewari, Sanghyun Park, and Sriram Padmanabhan. On space management in a dynamic edge data cache. In *Proc. Int. Workshop on the Web and Databases (WebDB)*, pages 37–42, Madison, Wisconsin, June 2002.

[AWY99] Charu Aggarwal, Joel L. Wolf, and Philip S. Yu. Caching on the World Wide Web. Knowledge and Data Engineering, 11(1):94–107, 1999.

[BAK+03] Christof Bornhövd, Mehmet Altinel, Sailesh Krishnamurthy, C.Mohan, Hamid Pirahesh, and Berthold Reinwald. Dbcache: Middle-tier database caching for highly scalable e-business architectures. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 662–662, San Diego, California, June 2003.

[BGM92] Daniel Barbará and Hector Garcia-Molina. The demarcation protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *EDBT*, volume 580 of Lecture Notes in Computer Science, pages 373–388. Springer, 1992.

[BPK02] Julie Basu, Meikel Poess, and Arthur M. Keller. Performance Analysis of an Associative Caching Scheme for Client-Server Databases. Technical report, Stanford Computer Science Technical Note:STAN-CS-TN-97-55, 2002.

[BR02] Laura Bright and Louiqa Raschid. Using Latency-Recency Profiles for Data Delivery on the Web. In *Proc. Int. Conf. On Very Large Data Bases (VLDB),* pages 550–561, Hong Kong, China, August 2002.

[CAL+02] K. Selçuk Candan, Divyakant Agrawal, Wen-Syan Li, Oliver Po, and Wang-Pin Hsiung. View invalidation for dynamic content caching in multitiered architectures. In *Proc. Int. Conf. On Very Large Data Bases (VLDB),* pages 562–573, Hong Kong, China, August 2002.

[CB00] Boris Chidlovskii and Uwe M. Borghoff. Semantic Caching of Web Queries. *VLDB Journal - The International Journal on Very Large Data Bases*, 9(1):2–17, 2000.

[CDI99] Jim Challenger, Paul Dantzig, and Arun Iyengar. A Scalable System for Consistently Caching Dynamic Web Data. In *Proc. IEEE INFOCOM Conf. on Computer Communications*, New York, New York, 1999.

[CHS99] Francis Chu, Joseph Y. Halpern, and Praveen Seshadri. Least expected cost query optimization: An exercise in utility. In *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems(PODS)*, pages 138–147, Philadephia, PA, June 1999.

[CLL+01] K. Selçuk Candan, Wen-Syan Li, Qiong Luo, Wang-Pin Hsiung, and Divyakant Agrawal. Enabling dynamic content caching for database-driven web sites. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Santa Barbara, California, May 2001.

[Condor] Condor team. University of Wisconsin - Madison. http://www.cs.wisc.edu/condor/

[CS84] Michael J. Carey and Michael Stonebraker. The performance of concurrency control algorithms for database management systems. In *Proc. Int. Conf. on Very Large Data Bases (VLDB),* pages 107–118, San Francisco, CA, USA, 1984. Morgan Kaufmann Publishers Inc.

[DDT+01] Anindya Datta, Kaushik Dutta, Helen M. Thomas, Debra E. VanderMeer, Krithi Ramamritham, and Dan Fishman. A comparative study of alternative middle tier caching solutions to support dynamic web content acceleration. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 667–670, Roma, Italy, September 2001.

[DFJ+96] Shaul Dar, Michael J. Franklin, Björn T. J´onsson, Divesh Srivastava, and Michael Tani. Semantic data caching and replacement. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 330–341, Mumbai (Bombay),India, September 1996.

[DN00] Prasad Deshpande and Jeffrey F. Naughton. Aggregate aware caching for multi-dimensional queries. In *Proc. Int. Conf. On Extending Database Technology (EDBT)*, London, UK, 2000. Springer-Verlag.

[DR99] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top n queries. In Proc. *Int. Conf. on Very Large Data Bases (VLDB)*, pages 411–422, Edinburgh, Scotland, September 1999.

[DRSN98] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. Caching multidimensional queries using chunks. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 259–270, Seattle, Washington, June 1998.

[Fra96] Michael J. Franklin, Client data caching: A foundation for high performance object oriented database systems. Kluwer, 1996.

[Gal99] Avigdor Gal. Obsolescent materialized views in query processing of enterprise information systems. In *Proc. ACM CIKM Int. Conf. on Information and Knowledge Management*, pages 367–374, New York, NY, USA, 1999. ACM Press.

[GL01] Jonathan Goldstein and Per-Ǻke Larson. Optimizing queries using materialized views: a practical, scalable solution. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 331–342, Santa Barbara, California, May 2001.

[GLR05a] Hongfei Guo, Per-Ǻke Larson, and Raghu Ramakrishnan. Caching with "good enough" currency, consistency, and completeness. *Technical report, TR1520, University of Wisconsin, 2005*. http://cs.wisc.edu/ guo/publications/TR1450.pdf.

[GLR05b] Hongfei Guo, Per-Ǻke Larson, and Raghu Ramakrishnan. Caching with "good enough" currency, consistency, and completeness. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, Trondheim, Norway, 2005.

[GLRG04] Hongfei Guo, Per-Ǻke Larson, Raghu Ramakrishnan, and Jonathan Goldstein. Relaxed currency and consistency: How to say "good enough" in sql. *In Proc.*

ACM SIGMOD Int. Conf. on Management of Data, pages 815–826, Paris, France, June 2004.

[GMW82] Hector Garcia-Molina and Gio Wiederhold. Read-only transactions in a distributed database. *ACM Transactions on Database Systems (TODS)*, 7(2):209–234, 1982.

[GN95] Rainer Gallersdörfer and Matthias Nicola. Improving performance in replicated databases through relaxed coherency. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 445–456, Zurich, Switzerland, September 1995.

[HB04] Theo Härder and Andreas Bühmann. Query processing in constraint-based database caches. *IEEE Data Engineering Bulletin*, 27(2), 2004.

[Hen04] Ken Henderson, The Guru's Guide to SQL Server Architecture and Internals, Addison Wesley, 2004.

[HSW94] Yixiu Huang, Robert H. Sloan, and Ouri Wolfson. Divergence Caching in Client Server Architectures. In *PDIS*, pages 131–139, Washington, DC, USA, 1994. IEEE Computer Society.

[Ise01] David Iseminger. Microsoft SQL Server 2000 Reference Library, Vol 4, Replication and English Query, Microsoft Press, 2001.

[KB96] Arthur M. Keller and Julie Basu. A predicate-based Caching Scheme for Client-server Database Architectures. In *VLDB Journal - The International Journal on Very Large Data Bases, 1996*.

[KFDA00] Donald Kossmann, Michael J. Franklin, Gerhard Drasch, and Wig Ag. Cache investment: integrating query optimization and distributed data placement. *ACM Trans. Database Systems.*, 25(4):517–558, 2000.

[KKST98] Anne-Marie Kermarrec, Ihor Kuz, Maarten Van Steen, and Andrew S. Tanenbaum. A framework for consistent, replicated web objects. In *Proc. Int. Conf. on Distributed Computing Systems(ICDCS)*, page 276, Washington, DC, USA, 1998. IEEE Computer Society.

[KR99] Yannis Kotidis and Nick Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 371–382, Philadelphia, PA, June 1999.

[LC99] Dongwon Lee andWesley W. Chu. Semantic caching via query matching for web sources. In *Proc. ACM CIKM Int. Conf. On Information and Knowledge Management*, pages 77–85, New York, NY, USA, 1999. ACM Press.

[LC02] Susan Weissman Lauzac and Panos K. Chrysanthis. Personalizing information gathering for mobile database clients. *In SAC,* March 2002.

[Len96] Richard Lenz. Adaptive distributed data management with weak consistent replicated data. In *Symposium on Applied Computing*, pages 178–185. ACM Press, 1996.

[LGGZ04] Per-Åke Larson, Jonathan Goldstein, Hongfei Guo, and Jingren Zhou. Mtcache: Mid-tier database cache in sql server. *IEEE Data Engineering Bulletin*, 27(2):35–40, 2004.

[LGZ03] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. Transparent mid-tier database caching in sql server. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 661–661, San Diego, California, USA, June 2003.

[LGZ04] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. Mtcache: Transparent mid-tier database caching in sql server. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 177–189, Washington, DC, USA, 2004. IEEE Computer Society.

[LKM+02] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 600–611, Madison, Wisconsin, June 2002.

[LN01] Qiong Luo and Jeffrey F. Naughton. Form-based proxy caching for database-backed web sites. In *Proc. Int. Conf. On Very Large Data Bases (VLDB)*, pages 191–200, Roma, Italy, September 2001.

[LR01] Alexandros Labrinidis and Nick Roussopoulos. Update Propagation Strategies for Improving the Quality of Data on the Web. In *VLDB Journal - The International Journal on Very Large Data Bases*, pages 391–400, 2001.

[LR03] Alexandros Labrinidis and Nick Roussopoulos. Balancing performance and data freshness in web database servers. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 393–404, Berlin, Germany, September 2003.

[Mesquite] Mesquite Software, “C++/CSim User’s Guide”. http://www.mesquite.com/

[Moh01] C. Mohan. Caching technologies for web applications. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, Roma, Italy, September 2001.

[OLW01] Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive Precision Setting for Cached Approximate Values. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 355–366, Santa Barbara, California, May 2001.

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 297–306, Washington, D.C., May 1993.

[OW00] Chris Olston and Jennifer Widom. Offering a precision performance tradeoff for aggregation queries over replicated data. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 144–155, Cairo, Egypt, September 2000.

[PL91] Calton Pu and Avraham Leff. Replica control in distributed systems: An asynchronous approach. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 377–386, Denver, Colorado, May 1991.

[RG02] Raghu Ramakrishnan and Johannes Gehrke. Database management systems. McGraw-Hill, Inc., New York, NY, USA, August 2002.

[RS77] Daniel R. Ries and Michael Stonebraker. Effects of locking granularity in a database management system. *ACM Trans. Database Systems.*, 2(3):233–246, 1977.

[RS79a] Daniel R. Ries and Michael R. Stonebraker. Locking granularity revisited. *ACM Transactions on Database Systems (TODS)*, 4(2):210–227, 1979.

[Sch86] Herb Schwetman. Csim: a c-based process-oriented simulation language. In *WSC '86: Proceedings of the 18th conference on Winter simulation*, pages 387–396, New York, NY, USA, 1986. ACM Press.

[SF90] Arie Segev and Weiping Fang. Currency-based updates to distributed materialized views. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 512–520. IEEE Computer Society, 1990.

[SJGP90] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, and Spyros Potamianos. On rules, procedures, caching and views in data base systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 281–290, Atlantic City, NJ, May 1990. ACM Press.

[SK97] Leonard J. Seligman and Larry Kerschberg. A mediator for approximate consistency: Supporting ”good enough” materialized views. *Intelligent Information Systems (JIIS)*, 8(3):203–225, 1997.

[SR90] Amit P. Sheth and Marek Rusinkiewicz. Management of interdependent data: Specifying dependency and consistency requirements. In *Workshop on the Management of Replicated Data*, pages 133–136, 1990.

[SS02] Andr´e Seifert and Marc H. Scholl. A multi-version cache replacement and prefetching policy for hybrid data delivery environments. In *Proc. Int. Conf. on*

Very Large Data Bases (VLDB), pages 850–861, Hong Kong, China, August 2002.

[SSV96] Peter Scheuermann, Junho Shim, and Radek Vingralek. Watchman : A data warehouse intelligent cache manager. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 51–62, Mumbai (Bombay),India, September 1996.

[SSV99] Junho Shim, Peter Scheuermann, and Radek Vingralek. Dynamic Caching of Query Results for Decision Support Systems. In *Scientific and Statistical Database Management Conference*, Washington, DC, USA, 1999. IEEE Computer Society.

[Tea02] Times-Ten Team. Mid-tier caching: the timesten approach. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 588–593, Madison, Wisconsin, June 2002.

[WQ87] Gio Wiederhold and Xiaolei Qian. Modeling asynchrony in distributed databases. In *Proc. Int. Conf. on Data Engineering (ICDE)*, pages 246–250. IEEE Computer Society, 1987.

[WQ90] GioWiederhold and Xiaolei Qian. Consistency control of replicated data in federated databases. In *Workshop on the Management of Replicated Data*, pages 130–132, 1990.

[WXCJ98] Ouri Wolfson, Bo Xu, Sam Chamberlain, and Liqin Jiang. Moving objects databases: Issues and solutions. In *Statistical and Scientific Database Management*, pages 111–122. IEEE Computer Society, 1998.

[YV00a] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pages 305–318, New York, NY, USA, 2000. ACM Press.

[YV00b] Haifeng Yu and Amin Vahdat. Efficient numerical error bounding for replicated network services. In *Proc. Int. Conf. On Very Large Data Bases (VLDB)*, pages 123–133, Cairo, Egypt, September 2000.

[ZLG05] Jingren Zhou, Per-Ǻke Larson, and Jonathan Goldstein. Partially materialized views. Technical report, MSR-TR-2005-77, Microsoft Research, 2005. ftp://ftp.research.microsoft.com/pub/tr/TR-2005-77.pdf.