Relaxed Currency and Consistency: How to Say "Good Enough" in SQL

Hongfei Guo
University of Wisconsin
guo@cs.wisc.edu

Per-Åke Larson
Microsoft
palarson@microsoft.com

Raghu Ramakrishnan
University of Wisconsin
raghu@cs.wisc.edu

Jonathan Goldstein
Microsoft
jongold@microsoft.com

ABSTRACT

Despite the widespread and growing use of asynchronous copies to improve scalability, performance and availability, this practice still lacks a firm semantic foundation. Applications are written with some understanding of which queries can use data that is not entirely current and which copies are "good enough"; however, there are neither explicit requirements nor guarantees. We propose to make this knowledge available to the DBMS through explicit currency and consistency (C&C) constraints in queries and develop techniques so the DBMS can guarantee that the constraints are satisfied. In this paper we describe our model for expressing C&C constraints, define their semantics, and propose SQL syntax. We explain how C&C constraints are enforced in MTCache, our prototype mid-tier database cache, 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 by 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 that is "good enough" for their purpose.

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. Today, such relaxed currency 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. For example, the system could return a warning to the application or use another data source.

Suppose an application queries a replicated table where the replication engine is configured to propagate updates every 30 seconds. The application is then implicitly stating that it is willing to accept data that is up to 30 seconds old. Suppose that replication is later reconfigured to propagate updates every 5 minutes. Is 5 minutes still within the application's currency requirements? For some queries 5-minute old data may be perfectly fine but for others it may not. The system cannot provide any assistance in finding the queries whose currency requirements are no longer met because it does not know what the requirements are.

Data currency requirements are implicitly expressed through the choice of data sources for queries. For example, if a query Q_1 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 Q_2 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.

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.

To this end, we have defined a model for relaxed currency and consistency (C&C) constraints, including proposed SQL syntax, and defined the semantics of such constraints. We also describe how support for C&C constraints is implemented in our mid-tier database cache prototype, in particular, integration with the optimizer and the use of dynamic plans to enforce data currency.

This work was motivated by several 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 sys-

SIGMOD 2004, June 13–18, 2004, Paris, France.

tem 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 paper is organized as follows. In Section 2 we describe how to specify C&C constraints. Section 3 illustrates how support for C&C constraints is implemented in our prototype mid-tier database cache. We report some experimental and analytical results in Section 4. Finally, Section 5 presents conclusions and addresses future work. The semantics of C&C constraints is formally defined in the appendix.

2. 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 is described informally in this section; formal definitions are in the appendix.

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 BooksCopy that is refreshed once every hour. In this scenario, the currency of the BooksCopy is simply the elapsed time since this copy became stale (i.e., when the first update was committed to Books after the last refresh) to the commit time of the latest update transaction on the back-end database.

Suppose we have another replicated table, ReviewsCopy, that is also refreshed once every hour. The state of BooksCopy corresponds to some snapshot of the underlying database and similarly for ReviewsCopy. 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.

```
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

2.1 Single-Block Queries

To express C&C constraints we propose a new currency clause for SQL queries. The new clause occurs last in 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 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 states of the two input tables must be consistent, that is, be 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 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 Review 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

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 Section 5), 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 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 our paper.

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 paper.

2.2 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 sale of a book and consider the query Q2 in Figure 2.2. Note that such queries can arise in a variety of ways. For instance, in this case, 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 (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, 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".

Q2:

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

Q3:

```
SELECT   ...
FROM     Books B, Reviews R
WHERE    R.isbn = R.isbn
AND      B.isbn IN
  (
  SELECT   isbn
  FROM     Sales S
  WHERE    year='2003'
  CURRENCY BOUND
           10 min ON (S, B)
  )
CURRENCY BOUND
           10 min ON (B, R)
```

Figure 2.2: Multi-block example C&C constraints

2.3 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 and row R is updated and the change committed 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_2; 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 the first query. 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.

3. IMPLEMENTATION

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 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. 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 part locally and part 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 the 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 plan includes runtime checking of the currency of each local view used. Depending on the outcome of this check, the plan switches between using the local view or 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 supports only table-level consistency and does not allow C&C constraints with grouping columns, such as 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 because views are updated by transactional replication.

We rely on a new SwitchUnion operator that has recently been added to SQL Server. A SwitchUnion 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 among the first N inputs to use. We use a SwitchUnion operator to transparently switch between retrieving data from a local view and retrieving it by 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.

3.1 Currency 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 currency 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 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 currency region.

Our current prototype is somewhat simplified and does not implement currency 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 currency 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 currency region. The table has two columns: a currency 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 currency regions, but this precludes having different heartbeat rates for different regions.)

Each currency region replicates its row from the heartbeat table into a local heartbeat table for the region. The agent corresponding to the currency 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 $t - T$.

3.2 Enforcing C&C Constraints

How can we efficiently ensure that a query result meets the stated C&C requirements? Our approach is to enforce consistency constraints at optimization time and at runtime enforce currency constraints. This approach requires re-optimization only if a view's consistency properties change.

3.2.1 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, $C = \{<b_1, \mathbf{S_1}>, \ldots, \{<b_n, \mathbf{S_n}>\}$, 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, $\mathbf{S_1}$>,…, {<b_n, $\mathbf{S_n}$>} is in normalized form if all input operands (in the sets $\mathbf{S_i}$) are base tables and the input operand sets $\mathbf{S_1}$,…, $\mathbf{S_n}$ are all non-overlapping.

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.

We briefly outline how to transform a set of constraints into normalized form but omit the actual algorithm due to lack of space. First, the algorithm recursively expands all references to views into references to base tables. Next, 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. If a query does not specify any currency clause, 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.

3.2.2 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, alternative algebraic expressions. 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 input a required sort property (a sequence 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.

Required consistency 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.

Delivered consistency property

A delivered consistency property consists of a set of tuples {<R_i, $\mathbf{S_i}$>} where R_i is the id of a currency 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. We briefly outline the algorithm ideas but do not include the actual algorithms because of lack of space.

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 currency 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.

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 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 currency 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.

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 is based on this observation.

Consistency 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 currency 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 $\mathbf{S}_i \cap \mathbf{S1}_j \neq \emptyset$ 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 <Rd, $\mathbf{S_d}$ > 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 currency 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.

3.2.3 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. We use the SwitchUnion operator described earlier for this purpose.

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 produces a new substitute exploiting the view. More details about the view matching algorithm can be found in [GL01].

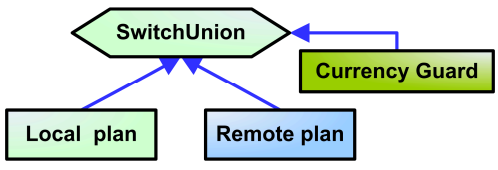


Figure 3.1: Substitute with SwitchUnion and a currency guard

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. 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 generated at the leaf-level but they can always be propagated upwards and adjacent SwitchUnion operators can be merged. 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.

3.2.4 Cost estimation

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

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

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 3.2. 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

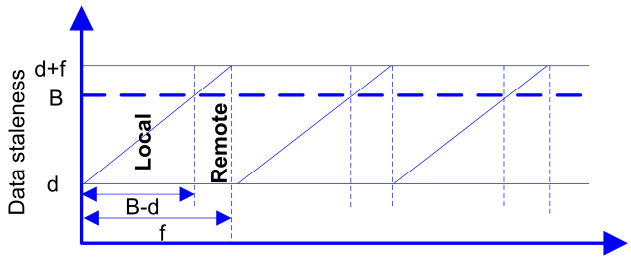


Figure 3.2: Sync cycle and data currency

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 chose because the local data is always sufficiently fresh so $p=1$. In summary, here is the formula used for estimating p:

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

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. 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.

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 currency 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: Currency region settings

4.1 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 Table 4.2, 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 parameters values used and the logical plans generated are shown in Table 4.3 and Figure 4.1, respectively. The rightmost column in Table 4.3 indicates which plan was chosen for each query.

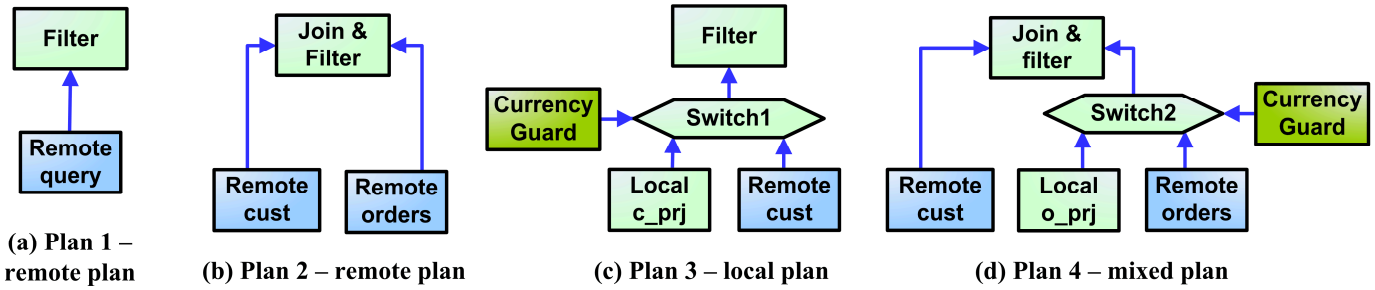
```
S1: SELECT   c_custkey, c_name, o_orderkey, o_totalprice
    FROM     customer, orders
    WHERE    c_custkey = o_custkey [AND c_custkey<$K]
    [CURRENCY clause]

S2: SELECT   c_custkey, c_name  FROM customer
    WHERE    c_acctbal between $A and $B
    CURRENCY BOUND 10 on (customer)
```

Table 4.2: Query schemas used for experiment

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

Table 4.3: Query variants used for experiment

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



(a) Plan 1 – remote plan **(b) Plan 2 – remote plan** **(c) Plan 3 – local plan** **(d) Plan 4 – mixed plan**

Figure 4.1: Generated logical plans — (a)-(d)

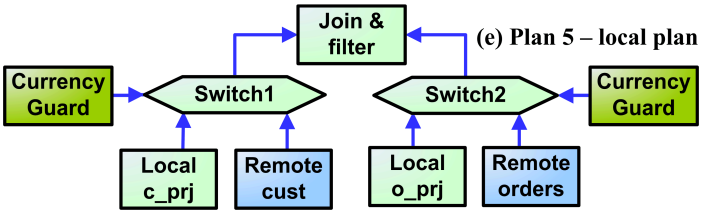


Figure 4.1: Generated logical plans — (e)

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 currency 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 became 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 4.1, 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.

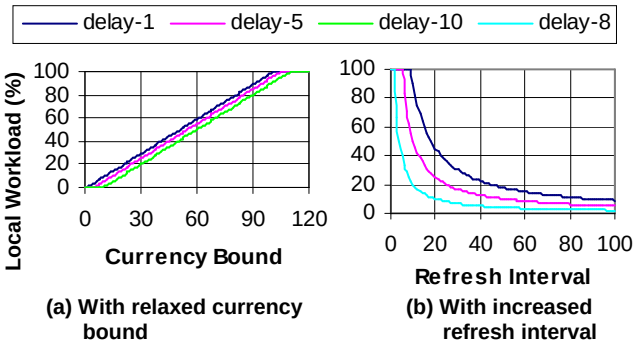


Figure 4.2: Workload shift

4.2 Workload Distribution

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 (previous section).

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*?

We plotted function (1) from Section 3.2.4 in Figure 4.2. In Figure 4.2(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 4.2(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.

4.3 Overhead of Currency Guards

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 4.4.

Q1 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 Q2, both servers used the same plan: a nested loop join with orders_prj (Orders) as the (indexed) inner. Again, for Q3, both servers used the same plan: a complete table scan.

Q1: key select	SELECT c_custkey, c_name, c_nationkey, c_acctbal, c_mktsegment FROM customer WHERE c_custkey = 1 [CURRENCY 10 on (customer)]
Q2: 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)]
Q3: 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 4.4: Queries used for experiment

	Local			Remote		
	Q1	Q2	Q3	Q1	Q2	Q3
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
# Rows	1	6	5975	1	6	5975

Table 4.4: Overhead of currency guards

	setup		run		shutdown		IdealTotal	
	ms	%	ms	%	ms	%	ms	%
Q1	0.04	27.13	0.06	152.52	0.01	26.56	~0.07	~11.51
Q2	0.06	39.39	0.09	98.52	0.01	29.69	~0.10	~14.32
Q3	0.01	2.98	1.99	3.79	0.04	46.21	~0.10	~0.16

Table 4.5: Local currency guards 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 Q1 and Q2 local execution, 1000 for Q3 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). Table 4.4 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 Q1 and Q2. 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 Q1 and 21% for Q2 seems surprisingly high, even taking into account that their very short execution time.

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 4.5 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 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 had 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 increase with the number of currency guards in the plan. For small queries such as Q1 and Q2, 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 SwitchUnions 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 Q1 and Q2 is relatively high because running the local plans is so cheap (Single indexed row retrieval for Q1, and 6-row indexed nested loop join for Q2). 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. Q1 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 Q3, 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 guards to the overhead in Q1 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 4.5.

5. RELATED WORK

Tradeoffs between data freshness and availability, concurrency and maintenance costs 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.

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 the freshness of the data? Different studies have tackled different aspects of this problem.

Quasi-copies [ABG88] 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 scenario 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 [WXC98] 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.

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 divergence bound guaranteed. Read-only Transactions [GW82], the Demarcation Protocol [BG92] and TACC [YV00] fall into this category. None of them supports queries with divergence bound constraints.

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.

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 the use of it 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, resulting in possibly higher overhead. Last but not least, enforcing currency requirements at the middleware level instead of inside the DBMS, FAS cannot provide transactional currency bound guarantees on query results.

6. CONCLUSIONS AND FUTURE WORK

This work was motivated by the lack of a rigorous foundation for the widespread practice of applications using replicated and cached data. To remedy the situation we proposed to allow applications to explicitly specify currency and consistency constraints in queries and have the DBMS enforce the constraints. We developed a model for C&C constraints and defined their semantics rigorously, thereby providing correctness standards for the use of replicated and cached data. We showed how C&C constraints can be expressed succinctly in SQL through a new currency clause. We described how support for C&C constraints is implemented in our prototype mid-tier database cache. C&C constraints are fully integrated into query optimization and execution.

This work provides a solid semantic foundation for the use of replicated and cached data but it is by no means complete; it can be extended in several directions. Regarding implementation, we plan to support timeline consistency and also finer-granularity consistency constraints, at the level of groups of rows. In the area of cache management, C&C constraints add more dimensions to this problem: even in the case of a cache hit, the local data might not be used simply because it does not satisfy consistency or currency constraints. We hope to develop caching mechanisms and policies that take these additional dimensions into account.

7. REFERENCES

[ABG88] R. Alonso, D. Barbará, H. Garcia-Molina, and S. Abad. Quasi-copies: Efficient Data Sharing For Information Retrieval Systems. In *EDBT*, 1988.

[ABK+03] M. Altinel, C. Bornhövd, S. Krishnamurthy, C.Mohan, H. Pirahesh, and B. Reinwald. Cache Tables: Paving The Way For An Adaptive Database Cache. In *VLDB*, 2003.

[BAK+03] C. Bornhövd, M. Altinel, S. Krishnamurthy, C.Mohan, H. Pirahesh, and B. Reinwald. DBCache: Middle-Tier Database Caching For Highly Scalable E-Business Architectures. In *SIGMOD*, 2003.

[BGM92] D.Barbará and H.. Garcia-Molina. The Demarcation Protocol: A Technique For Maintaining Linear Arithmetic Constraints In Distributed Database Systems. In *EDBT*, 1992.

[BR02] L. Bright and L. Raschid. Using Latency-Recency Profiles for Data Delivery on the Web. In Proc. In *VLDB*, 2002.

[CHS99] F. Chu, J. Halpern, and P. Seshadri. Least Expected Cost Query Optimization: An Exercise In Utility. In *PODS*, 1999.

[DR99] D. Donjerkovic and R. Ramakrishnan. Probabilistic Optimization Of Top N Queries. In *VLDB*, 1999.

[Gal99] A. Gal. Obsolescent Materialized Views in Query Processing of Enterprise Information Systems. In CIKM, 1999.

[GMW82] H. Garcia-Molina and G. Wiederhold. Read-Only Transactions In A Distributed Database. In *TODS*, 1982.

[GN95] R. Gallersdörfer and M. Nicola. Improving Performance In Replicated Databases Through Relaxed Coherency. In *VLDB*, 1995.

[GL01] J. Goldstein and P. Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *SIGMOD*, 2001.

[HSW94] Y. Huang, R. Sloan, and O. Wolfson. Divergence Caching in Client Server Architectures. In *PDIS*, 1994.

[KKST98] A. Kermarrec, I. Kuz, M. Steen, and A. Tanenbaum. A Framework For Consistent, Replicated Web Objects. In ICDCS, 1998.

[LC02] S.Weissman L. and P. Chrysanthis. Personalizing Information Gathering For Mobile Database Clients. In SAC, 2002.

[LGZ04] P. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent Mid-Tier Database Caching In Sql Server. In *ICDE*, 2004.

[LR03] A. Labrinidis and N. Roussopoulos. Balancing Performance And Data Freshness In Web Database Servers. In *VLDB*, 2003.

[OLW01] C. Olston, B. Loo, and J. Widom. Adaptive Precision Setting for Cached Approximate Values. In *SIGMOD*, 2001.

[OW00] C. Olston and J. Widom. Offering A Precision-Performance Tradeoff For Aggregation Queries Over Replicated Data. In *VLDB*, 2000.

[PL91] C. Pu and A. Leff. Replica Control In Distributed Systems: An Asynchronous Approach. In *SIGMOD*, 1991.

[RBSS02] U. Röhm, K. Böhm, H. Schek, and H. Schuldt. FAS - a Freshness-Sensitive Coordination Middleware for a Cluster of OLAP Components. In *VLDB*, 2002.

[SF90] A. Segev and W. Fang. Currency-based Updates To Distributed Materialized Views. In *ICDE*, 1990.

[SK97] L. Seligman and L. Kerschberg. A Mediator For Approximate Consistency: Supporting "Good Enough" Materialized Views. In *JIIS*, 8(3):203--225, 1997.

[SR90] A. Sheth and M. Rusinkiewicz. Management Of Interdependent Data: Specifying Dependency And Consistency Requirements. In *Workshop on the Management of ReplicatedData*, pages 133--136, 1990.

[WQ87] G. Wiederhold and X. Qian. Modeling Asynchrony In Distributed Databases. In *ICDE*, 1987.

[WXCJ98] O.Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues And Solutions. *In Statistical And Scientific Database Management*, pages 111--122, 1998.

[YV00] H. Yu and A. Vahdat. Efficient Numerical Error Bounding For Replicated Network Services. In *VLDB*, 2000.

8. APPENDIX: C&C SEMANTICS

8.1 Database Model

A **database** is modeled as a collection of **database objects** organized into one or more tables. The granularity of an object may be a view, a table, a column, a row or even a single cell in a row. Every object has a **master** and zero or more **copies**. The collection of all master objects is called the **master database**. A **cache** is any collection of copies, and can include (parts of) one or more database tables or materialized views.

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.

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. We denote the **history** after n update transactions have committed by H_n, and usually treat it as $H_n = T_1 \circ T_2 \circ \ldots \circ T_n$. A database state produced by applying a history H_n on the initial state is called a **snapshot** of the database. Since each update transaction maps one-to-one with a timestamp, as well as with a snapshot of the master database, we sometimes use these concepts (update transaction T_n, history H_n, snapshot of the master database) interchangeably.

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.

Transaction Timestamps: The function **xtime**(A) returns the transaction timestamp of transaction A. We overload the function **xtime** to apply to objects. The transaction timestamp associated with a master copy O, **xtime**(O, H_n), is equal to **xtime**(A), where A is the latest transaction in H_n that modified O. For a copy C, the transaction timestamp **xtime**(C, H_n), is copied from the master object by the DBMS when the copy is synchronized, i.e., when a special copy-transaction copies the value of the master to the slave object. Note that if C is a copy of master object O, then **xtime**(O, H_n) ≥ **xtime**(C, H_n).

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

Copy Staleness (Currency): Given a database snapshot H_n, a copy C is stale if **master**(C) is modified in H_n after **xtime**(C, H_n). The time when C became stale, called the *stale point*, **stale**(C, H_n), is equal to **xtime**(A), where A is the first transaction that modifies **master**(C) after **xtime**(C, H_n) in H_n. If C is not stale in H_n **stale**(C, H_n) is defined to be **xtime**(T_n).

8.2 The Extended Query

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 version Q^{ext}. The construction proceeds block-at-a-time, and ensures that the result of Q^{ext} includes all objects used in constructing the result of Q (including objects used in testing Where clauses, grouping, etc.). We refer to the result of Q^{ext} as the **relevant set** for Q. (We omit the details of the construction for lack of space.)

8.3 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, using comparison operators, quantifiers and Boolean connectives.

8.4 Currency Constraints

For a query Q, a user can specify currency requirements for any copy C in the complete extended query set $Q^{ext-all}$ by comparing C with its counterpart in the master copy of the results of $Q^{ext-all}$, 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).

8.5 Group Consistency of 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 regards to H_n, then we say K is **snapshot consistent** with respect to H_n. If K is the entire cache, we say 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 8.1 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 objects K cached objects in 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.

8.6 Group Consistency for Queries

Our approach to consistency constraints in a query specification reflects two principles:

1) Consistency 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 Q^{ext}.

2) It must be possible to require consistency for subsets of the data used in a query; we achieve this, naturally, by leveraging the query mechanism to identify the subsets.

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

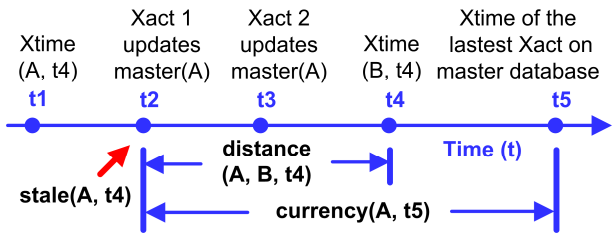


Figure 8.1: Basic concepts

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.

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.

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).

8.7 Inter-Group Consistency

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

- Multiple groups of cached objects, such as all cached Order records and all cached Catalog records.
- Groups of objects from different blocks of a query. (Observe that each subquery has an extended version!)
- 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 ∈ Gi, B ∈ Gj, **xtime**(A, H_n) $\leq$ **xtime**(B, H_n), where H_n is the database snapshot after executing all statements corresponding to the groups G1, G2, … , Gn.