Mass Collaboration: A Case Study

Raghu Ramakrishnan
University of Wisconsin-Madison, raghu@cs.wisc.edu

Andrew Baptist
ADP, abaptist@cs.wisc.edu

Vuk Ercegovac
University of Wisconsin-Madison, vuk@cs.wisc.edu

Matt Hanselman
ADP, mjhans@cs.wisc.edu

Navin Kabra
Veritas, navin.kabra@veritas.com

Amit Marathe
AT&T Research, marathe@research.att.com

Uri Shaft
Oracle Corporation, uri.shaft@us.oracle.com

Abstract

We present an overview of a novel customer support system developed at QUIQ between 1999 and 2003. The application was perhaps the first systematic use of mass collaboration, *which builds on the observation that large communities of users can be effectively leveraged to help each other, and to advance the interests of the community as a whole. In recent years, mass collaboration has been proposed for uses such as information integration and even program debugging, and shows much promise. In this paper, we outline the main ideas and technical challenges, and describe the QUIQ architecture. Technically, the main achievements include a novel DB-IR engine; a scalable notification engine for a rich class of user-specified alerts; a powerful access control mechanism with support for roles, dynamic groups, and field-level access control; and techniques for editing navigation hierarchies in dynamic sites.*

1. Introduction

1.1. Mass Collaboration for Customer Support

Customer-support plays a key role in retaining and expanding a company's customer base, and companies typically offer many channels of support, including phone and online support. Online support, especially self-service through a knowledge-base, is by far the least expensive channel, provided that the customer can find a satisfactory answer easily. If a question escalates to a phone call, and then to a "ticket" in a request-tracking system, the cost of resolving the issue escalates accordingly (and rapidly). For many settings, unfortunately, it is impractical to maintain a comprehensive and current knowledge-base, for a variety of reasons—sometimes, an issue involves several products, some from other vendors, in unexpected combinations; or the product line may be evolving quickly, with many versions, each with their own quirks. Obviously, such scenarios also make it difficult to train support personnel.

The established base of customers who use the product, not surprisingly, is often the best place to look for knowledgeable experts. And perhaps surprisingly, there are typically a large number of customers who are willing to help other customers out by answering a question. The idea behind the QUIQ customer-support application, used by a number of companies in the high-tech sector, is to tap into this community of customers as a source of support. The simplest approach is to simply use a message-board, and to let customers discuss issues with other customers. However, message-boards are designed for casual threads of discussion, not for goal-directed interactions. They have poor search capabilities, no mechanism for role-based control of information flow, making it difficult to seamlessly integrate multiple levels of data and service based on the level of support that different groups of customers have contracted for. The QUIQ system was designed to address these limitations, while retaining a simple message-board type interface to encourage casual interactions, and has the following features:

- **User-Centric Organization:** The user interface was designed to help users find answers to their questions by either browsing hierarchically organized web pages or posting their question. When posting a question, a user is presented with a set of answers, and proceeds to post only if there is no satisfactory answer. Thus, assuming that the search capability can retrieve appropriate answers if they exist, the same issue is not addressed over and over, as in a message board; rather, an existing dialog might be deepened and commented upon further. What a user sees at any point is governed by a number of factors, including their customer-support level and the corresponding access to various internal knowledge-bases, and the user groups they belong to. However, the user never has to keep these nuances in mind, and can simply ask a question; the system takes the user's privileges into account and returns the best answers that the user is authorized to see.

- **Routing Services:** Once a question is asked, several routing mechanisms exist to ensure that a satisfactory answer is returned. First, users can specify sophisticated *saved searches*, and receive notifications by email when relevant answers are posted. Second, posted questions enter a workflow, and are escalated to the attention of paid support-personnel (perhaps by logging them into another CRM application, such as Siebel) if they are not answered (by the community at large) within the window of time appropriate to the customer's support level. Observe that posted messages should be seen in real time. (A user who posts a question will be annoyed if it is not immediately visible! Indeed, it must be searchable in real-time, since we want timely answers from other viewers, within short contractually-specified timeframes.) Posted answers also go through a customizable workflow, to ensure desired levels of quality. For example, answers posted by a typical customer are visible immediately, while those posted by 'experts' (either distinguished customers invited to this status, or paid support personnel) might require a round of editing and approval.

- **Intelligent Search:** In terms of both presentation and internal structure, all posted messages are organized in question-centric units to facilitate search. These units consist of a question and all answers posted in response, together with a thread of discussion per answer. When searching, this unit is the analog of a 'document', and the unit of retrieval. Matches are given different importance based on whether the search terms appear in the question part, an answer, or in a subsidiary discussion. In addition to messages posted by users, each question-unit has additional 'hidden' text and metadata, which is also used when searching for matches to a customer's question. In addition to information such as the poster's identity and the time of posting, information about the poster's *authority*, the *quality* of an answer, the *popularity* of a question, and a *profile of related searches* are maintained. (See Section 5.) Typical user searches are translated into complex selection queries with twenty to thirty constraints, including about a half-dozen keyword constraints. Results are ranked, and must reflect recent updates in near real-time.

- **Data Mining and Business Intelligence:** User activity in the form of postings and searches is a rich source of information about individuals and postings, in terms of their influence and value in the community. Mining this content can yield detailed user and content profiles, which are valuable for search and content routing, as noted earlier. The QUIQ system includes an automatically updated warehouse that integrates information from the database as well as the saved searches engine and web browser logs, which are carefully instrumented to provide detailed information such as context and duration for each click. Continually refreshing the database indexes to reflect these constantly updated profiles is essential for searches to fully benefit from this information, and is achieved as part of the novel hybrid DB-IR engine. Further, the warehouse is the basis for extensive reporting capabilities, and supports functionality such as identifying active participants who are candidates for recognition and incentives.

The rest of this paper is organized as follows. We describe the overall Architecture in Section 2. We discuss the role- and group-based authorization capability in Section 3. We outline the Token Index in Section 4, and the saved searches engine in Section 5. A novel aspect of the QUIQ system is its extensive use of a new *hierarchy* data type; we describe this in Section 6.

2. System Architecture

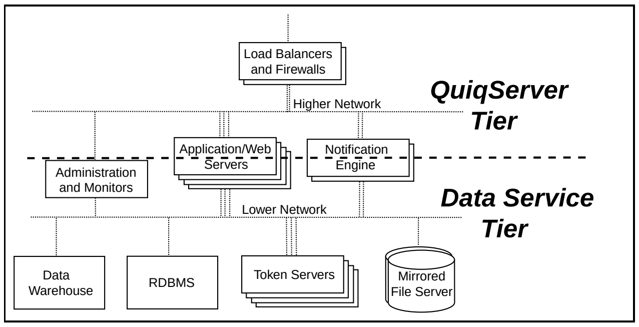

Figure 1. QUIQ System Architecture.

The QUIQ architecture is divided into two tiers, as shown in Figure 1:

Data Service Tier: This tier is responsible for storage and retrieval of the data. This includes both the knowledgebase (containing information about users and their questions/answers) and the data warehouse.

QUIQServer Tier: An application server that acts as a bridge between the end-user and Data Service Tier.

2.1. The Data Service Tier

The components of this tier are as follows:

Relational Database Management System (RDBMS): We use a commercial database system to handle the ACID transaction properties required by the whole system. The database is also used for communication between components when we need to guarantee ACID transaction properties on the communication. We also use the database to log jobs to be performed by the *token index* and the SSE.

Token Index: This component [13, 14] handles queries that involve some information retrieval. It handles queries of the kind “Find all tuple-ids in a given table whose fields contain a specific list of tokens.” The token index can handle boolean combinations of constraints or the form “token T exists in field F”. The index is updated through a job table in the database. It ensures a very short delay (seconds) between the time an update is posted in the job table and the time it is indexed and can be fetched as a query result.

File System: The file system is used for fast storage, update and retrieval for large documents. It is much more efficient than database Large Objects (LOBs). We maintain ACID transaction properties over the file system using a technique that is almost identical IBM’s method [1]. (We implemented this part of the system independently about three years before that publication.)

Saved Search Engine (SSE): Notification requests, or *saved searaches*, are maintained in the database and can be managed by the user through simple application work-flows. The SSE builds an index on all these notification requests. It polls all tables in the database on a regular schedule (using timestamps) and combines the results into notification emails.

Data Warehouse: This is another commercial relational database system used to support data mining and reporting. The data is gathered by the administration machines from all sources (RDBMS, file system, logs from the application servers).

Connection Software: We embedded the software for connecting all the Data Service Tier components in the QUIQServer application tier software. The most important part of the QUIQServer software is the Query Optimizer. Since some of the indexes in the system reside in the Token Index machines, and some data (documents) resides outside the database, we need a query language and query optimizer to bind these resources together.

2.2. The QUIQServer Tier

This tier is responsible for managing a request from the user, such as a search or posting of a question. The two main components are the Application Server and the Saved Search Engine (SSE). The block diagram in Figure 1 shows these components as belonging to both the Data Service Tier and QUIQServer. The lowest layer of the software running in the Application Server and the SSE deals with access to the Data Service tier, and is essentially the call interface that knows how to interact with a relational database and a token index. The architecture of the SSE is discussed in Section 5. The rest of this section covers the Application Server, which contains three layers: *Data Access*, *Business Logic* and *Presentation*.

Data Access: This layer manages the data schema for a QUIQ application. The schemas consist of tables (and indexes) using some common data types such as number, string and timestamp. The QUIQ application also uses the more complex data types *Hierarchy* and *Document*. In all cases, the data access layer translates common operations into the languages understood by the database, the token index and the file server. These operations include inserts, deletes, updates, queries, and schema modification. For queries, this layer is responsible for optimization when we need to access more than just the relational database. The schemas can specify indexes in both the relational database and the token index. Query optimization takes both types of indexes into consideration, and is discussed further in Section 2.3.

Business Logic: This layer is responsible for workflows of the application. We maintain a persistent user state between user requests. This state includes the most recent operations and the user’s credentials. This layer is responsible for translating a user request into operations that are given to the Data Access layer. For example, posting a question involves decoding the HTTP messages from the user, understanding which part of the workflow the user is in, and generating the appropriate queries and updates for the Data Access Layer.

Presentation Layer: This layer is responsible for creating a response to the user after the Business Layer concluded its job and issued a “commit” operation. The data accumulated by the Business Layer is transformed into a document that is returned to the user. This is usually an HTML web page. Some workflow results are plain text or XML.

2.3. Query Types and Optimization

Results to almost all queries initiated by users of a QUIQServer application are displayed to the user and not consumed by another application. This means that results are browsed in small quantities, and must be returned within a few seconds at most. We display the results in "windows", beginning with the top n results. When the user requests more, we display the next n results and so on. (The user can also go back to previous result windows.) We sort query results on a unique key, and get the next window of results by re-issuing the query with an additional constraint that the sort columns must be higher than the sort value of the previous window's last result.

QUIQServer supports three types of queries: *pure database query, pure token index query* and *hybrid* queries.

Pure Database Queries: These queries are processed by the relational database. Obviously, we are restricted to relational operators (i.e., standard SQL). To get the proper performance for result windows, we have to build the appropriate indexes and perform top n optimization, but most of the optimization can be left to the database system.

Pure Token index Queries: These types of queries select tuples from a single table using constraints that can all be evaluated by the token index. The results are returned in order of relevance (determined by our IR algorithm) and tuple-id (for uniqueness). Optimizations for result windows are built into the token index. The token index returns only relevance values and tuple-ids, so we need to go to the relational database to fetch the documents in the result window, based on the tuple-ids. Note that the token index can handle relational constraints on numbers, strings and even timestamps. Therefore, this query class is very useful.

Hybrid Queries: These queries require use of both the token index and the relational database. The query optimizer creates a query plan which is a tree of operators. There are only two types of leaf operators: relational query, and pure token index query. The connecting operators can perform sorting, joins, filtering, etc. In this case, most of the optimization is done by the QUIQServer software. In addition, we support sort orders that do not include the token index relevance calculation. A QUIQServer application will often return results sorted by some timestamp, or other relational column.

2.4. Deployment

The QUIQ architecture is designed for very high fault tolerance, with extensive use of parallelism to maintain high availability and scalability. A single deployment consists of multiple application server machines, multiple token indexes and a mirrored file server. In addition, the QUIQ architecture includes monitoring machines that check on all the other components (and each other) and notify the administrators of any problems. These monitors can also restart failing components automatically. The QUIQ architecture is designed as a hosted solution, which allows a continuum of sharing of computational resources amongst customers.

3. Access Control

There are two levels of access control in the QUIQ system. The first level is authentication, and this is handled through a password mechanism. When a user logs in, the system loads and caches the *capabilities* for the user. These capabilities, described below, determine what the user can view and do (e.g., in terms of various system workflows), and are enforced in two steps: (1) Every action in the system involves issuing a query (if only for authorization purposes), and the current user's capabilities are automatically expanded into additional query constraints. (2) The QUIQServer query evaluation enforces these constraints.

3.1. Example

Consider the scenario where the QUIQ Application is used in the context of a photo editing application. Questions are organized by a hierarchy of issues. For example, Alice frequently reads and posts questions related to *Installation* issues. Bob on the other-hand is an expert in the *Edge-detector* plug-in which is found under the generic *Plug-in* category. We would like to specify that Alice is an enthusiastic user whose contributions are valued and is restricted to post no more than five questions to the installation category. Bob is an expert so can contribute ten questions. In addition, questions may be marked as requiring expert assistance thus routing them to users such as Bob.

Access control in the QUIQ architecture allows such rules to be easily specified by using a *Role* based system. A dynamic set of roles is supported and for each role, a dynamic set of users may be associated. From the example above, we have two roles: *expert* and *enthusiast*. Each role can be used to determine whether a question can be read and how many postings are allowed. This illustrates how roles can be parameterized. Additionally, membership in a role can also be parameterized. For example, Bob's role as an expert is restricted to the *plug-in/edge-detector* category while Alice's enthusiasm is similarly restricted to *installation*. The following sections describe how this example can be enforced.

3.2. Access Control Structures

Two tables are used to describe access control in a QUIQ application:

Capability: A set of roles and for each role, the parameterization of each capability supported by the system. A capability parameterization can be used to restrict access to data or determine how an action is evaluated. The key for Capability is <Role>; its schema is as follows:

<Role, cap1, cap2, ... capN>

Membership: An association between users and roles defined in the Capability table. Each association can be parameterized to restrict the portion of the database over which a rule is applicable on a per user basis. The key for Membership is <UserId, Role>; its schema is as follows:

<UserId, Role, Param1, Param2, ... ParamN>

From the example above, the following tables would be used:

Role	Private?	Limit
Expert	true	10
Enthusiast	false	5

Table 1. Example of Capability Table

UserId	Role	Category
Alice	Enthusiast	Installation
Bob	Expert	Plug-in/Edge Detector

Table 2. Example of Membership Table

The set of all capabilities in the system is cached across all the servers at initialization time. In addition the set of memberships that a user has is loaded when the user first logs in and stored in the session. The computation of whether the user has permission to perform an operation is calculated at run time from the two in memory caches using the combination functions defined above. The caching at the user level is periodically refreshed (every 15 minutes) in order to handle any changes that may occur to the users permission while accessing the system.

3.3. Access Control for Queries

In the case of queries, the capabilities associated with the user who submits the query are used to *rewrite* the query such that only the accessible records are returned. From the example, if Alice requests questions from either the Installation, Plug-in, or Edge-detector categories, the resulting questions will only contain *public* questions. The same is true for Bob except for the Edge-detector category where *private* questions will also be visible.

The query example illustrates row-based access control. If access control over a record's field value is required, the same data structures are used in a post-processing step that follows the retrieval of records from the data servers and preceeds the presentation to the user.

3.4. Access Control for Actions

For actions such as posting *questions*, the same access control data-structures are used. The limit of five postings for Alice in the Installation category may be more than other categories, allowing her posting to succeed. Similarly, Bob's posting to Plug-in may not succeed depending on his current posting count for that category whereas for Edge-detector, the posting may succeed.

Because a user can be a member of several roles and they can have conflicting capabilites, there is a need to resolve these conflicts. For the simple boolean capability, there are 2 options for resolution, either the permission is granted if ANY of the roles has the capability, or the permission is granted only if ALL the roles have the capability. Typically the ANY combination would be used to grant additional permissions while the ALL combination could be used to restrict permissions. For more complex types any function can be applied to return a single value based on a set of input capbilities and memberships.

4. Token Index

The QUIQ architecture bridges relational queries and IR search: (1) Relational and keyword predicates can be used to filter records, and (2) a scoring function that determines relevance can be computed based on both relational and free-text attributes. Such a query paradigm is supported by the QUIQServer query evaluation component, making use of the *Token Index*. The QUIQServer is responsible for building the query to send to the Token Index, merging the results with other data servers, and incorporating *transparency feedback* from the Token Index in order to better explain to the end user the reason for returning each result. The following sections discuss in greater detail the functionality supported, the implementation approaches taken, and the query paradigm.

4.1. Token Index Functionality

The Token Index is an inverted index that maps *tokens* from a given collection to *posting lists* consisting of a set of record identifiers (RID's) and a count. The count represents the number of occurences of a token within a *token-field*. A token-field may correspond directly to a collection field or may be the result of some function. For example, a token-field defined as the union of all text fields in a collection is maintained in the Token Index to facilitate queries over any record fields containing free-text. Any data type that can produce tokens can utilize the Token Index.

The operations supported by the Token Index are:

- **Query:** retrieve a set of RIDs computed through the evaluation of a query using the query paradigm described in section 4.3. The result may be sorted by score and a data structure aiding transparency can be optionally returned. The set can be retrieved in batches in order to increase responsiveness.
- **Modification:** updates, inserts, and deletes are supported at the record level.
- **Bulkload:** an algorithm optimized for loading large collections of data rather than freshness.

4.2. Token Index Implementation

The Token Index consists of two types of processes: **servers** maintain the inverted index and answer queries submitted by the QUIQServer while **readers** distribute to all servers the necessary updates in order to synchronize them with the current database state. Our basic idea is to *defer applying update operations* to the servers' persistent store. Updates are handled in three steps: (1) Changes to the database are reflected in a special table. (2) The changes are continually polled by the reader process and incorporated into a differential index structure. (3) The main index is periodically refreshed to absorb the differential index. These details must be transparent to data retrieval operations, and therefore retrieval operations have an additional step of checking results against the differential index to adjust for changes that have not yet made it to the main index.

The differential approach to managing updates is implemented through two types of indexes:

Static index is persistent and is organized exclusively for efficient retrieval. Its posting lists are tightly packed and possibly compressed on disk.

Dynamic index is transient and in-memory, and is organized for efficiently accomodating updates.

4.2.1. Operations: The format for insert and update operations is assumed to be record oriented; multiple token-fields with multiple tokens are provided per RID. Deletes only specify the RID. Queries are assumed to be an expression tree whose leaf-level nodes fetch a posting list given a token. The operations use the two-level index structure as follows:

Insert: If no entry is found for a token in the dynamic index, a new posting list is created containing the RID. If a posting list exists with the given RID, the count is incremented.

Update: A posting list for a token may contain entries for a RID in either the static or dynamic index. If a RID does not exist in the dynamic index, the operation proceedes as an insert and takes precedence for reads over the static entry. The complicated case arises when the dynamic index contains entries for the RID. In this case the most recent token-field state must be preserved in order to compute the appropriate token inserts as well as counter increments and decrements.

Delete: A bit-vector where the ith position determines whether a RID is deleted or not.

Read: The posting lists found in the static and dynamic indexes are *merged*. The merge operation for reads first removes those static index RIDs that are
given precedence by updates in the dynamic index. Then, RID counts in both are added and the final result is masked by the delete bit-vector.

4.2.2. Merge: The approach used to defer changes is based on the assumption that the number of changes will be small relative to the number of queries. Periodically, the static and dynamic indices are merged into a new static index which replaces the existing static index. Two techniques are used in order to reduce down-time during merge: (1) the index is partitioned by token into N partitions and (2) the merge is written to a copy which replaces the existing version when merge is complete. The first technique allows the merge of the entire index to proceed incrementally. The second technique increases freshness by allowing reads and updates to proceed concurrently with merge.

In practice, the number of partitions used is 100 to 200 and cycling through all partitions for merge is spaced out over a 24 hour period. Continual rewriting effectively partitions the index by time of updates: *old* changes are in the static index and the *new* changes are in the dynaic index. Additional benefits are gained by piggy-backing extra decisions and updates while rewriting the entire index: (1) statistics from analysis/mining of query and update traces can be used for system self-tuning at the storage level (2) formatting updates required by new software versions are easily phased in.

4.2.3. Incorporating the Results of Data Mining: The Merge scheme described above provides a convenient insertion point for the results of data mining, which, in principle, continuously update each row describing a question or user with improved profile information. The updates are never propagated to the database itself. Rather, corresponding changes are made to the *token index*, and these changes are piggy-backed onto the merge step. Thus, we side-step the problem of declustering the DBMS's index structures due to a large number of updates on indexed fields.

4.2.4. Concurrency Control: Handling concurrent reads and update operations requires only the use of short-term latches since the static index on disk is not modified by updates. Merges however require extra care for correctness and efficiency. In order to allow reads and updates to proceed during a merge on partition P_i, the dynamic index for a partition is frozen by exclusively locking P_i. The timestamp T of the most recently applied job is recorded for recovery purposes and a new dynamic index is created that accepts updates. After releaseing the lock, the merge proceeds to create a new static index timestamped with T. P_i is exclusively locked while the current static index is replaced with the new static index which concludes the merge for P_i. During merge, readers must consider the static index as well as the frozen and current dynamic index. Down-time is minimized due to the short period that the partition is exclusively locked as well as the ability for updates and reads to proceed concurrently with the merge process.

4.2.5. Recovery: The Token Index is recoverable through the use of a *redo* log maintained as a table in a comercial database. The QUIQServer submits update jobs to the Token Index by inserting timestamped jobs into the redo table. The reader process uses the timestamp to insure that all servers obtain all jobs in the most same order. The delay in practice between the QUIQServer subitting a job and the server obtaining the job is 30 seconds. On start-up or during a crash recovery, the servers obtain their minimum T_i amongst all partitions, and the reader uses the minimum amongst all servers to determine the point in the redo table from where to begin reading jobs.

4.3. Token Index Query Paradigm

The query paradigm supported by the Token Index allows the combination of database-style *exact* queries along with IR-style *approximate* queries. Both types of queries can be applied to either free-text or relational data types through the appropriate value tokenization. A query is a tree of exact and approximate *constraints* where constraints are combined using AND and OR boolean connectives. Additionally, constraints may be weighted in order to *boost* scores depending on which token-fields they range over. The output is a list of *scored* RIDs.

Evaluating a query proceeds by obtaining token posting lists for the leaf constraints and annotating all constraints with a computed score and a flag indicating whether the result is approximate or exact. The scoring function used is the commonly used TF/IDF scoring function used in IR systems [22]. We are not constrained to a particular scoring function however. Marking constraints as exact or approximate determines whether a RID is included, how to combine scores, and how to annotate the parent node. Exact constraints are similar to the SQL WHERE clause in that RID's are filtered. Approximate constraints add RID's and score the results. Scores originating from either exact or approxiate nodes are summed together to produce the score for the parent contraint.

In addition to the scored result set, the Token Index can provide useful feedback that can be used increase the *transparency* of the results by explaining to the user why the results returned match their query. Several pieces of information are available: (1) the query plan is annotated with scores, (2) matched token inverse document frequency (IDF), and (3) each records matching tokens allong with their term frequency (TF). The feedback information is used to highlight words in the matching results, construct summaries from large text fields, and suggest queries to find results similar to a given record. In order to not overwhelm the user, the decision for which words to highlight and summarize is driven by the scores avaialable in the feedback.

More details about the Token Index query paradigm, implementation, and performance can be found in [14].

5. Saved Searches Engine

The Saved Searches Engine (SSE) manages all subscriptions within the system. A *subscription* is a query over the underlying tables that triggers a notification whenever a tuple satisfying the query enters the database (due to either an insert or an update). Subscriptions have also been called continuous queries and alert profiles elsewhere in the literature. They can be thought of as having three main aspects: *query*, *periodicity*, and *lifetime*. The query is the condition to be satisfied by a new (or updated) tuple to trigger this subscription. The periodicity determines how often the subscription is triggered within a time window. It can range from instant (notification generated immediately) to daily or weekly. For subscriptions with non-instant periodicity, multiple alerts are coalesced into a single notification. The lifetime component governs how long the subscription is retained by the SSE. This parameter is used to purge subscriptions that are unlikely to be triggered, or are not required, in the future.

5.1. Query Language

The query component is a necessary and sufficient condition to trigger the subscription. It is a logical expression involving constants, field names and arithmetic/comparison/logical operators that evaluates to true or false on every tuple. Conceptually, the SSE evaluates all queries for every new tuple and triggers those subscriptions for which the evaluation returns true.

It should be pointed out that joins are not permitted in the query. Thus all field names in the query refer to fields of a single table.

5.2. Indexing

The SSE design requirements call for managing millions of subscriptions at an update volume of several thousand tuples per day. Therefore, it is infeasible to evaluate every query for each new tuple. Rather, given a new tuple we should be able to quickly find all the matching queries (i.e. queries which evaluate to true on this tuple). To achieve this goal we build an index on the queries. This is the opposite of a conventional index which, given a query, enables fast retrieval of all matching tuples.

The index can be thought of as a map from atomic constraints to a bitmap whose length equals the number of queries in the system (an atomic constraint is an expression of the form "FIELDNAME = LITERAL"). The bitmap corresponding to atomic constraint c will have a 1 in position i if and only if query i directly or indirectly contains c. Therefore any new query the server encounters is broken down to extract a list of atomic constraints and this map (from atomic constraints to bitmaps) is updated according to the above rule.

The index also maintains for each query a triplet (min, exact, max) of integers (min $\leq$ exact $\leq$ max) with the following semantics:

min at least these many atomic constraints in the query have to be satisfied by a matching tuple.

exact if these many atomic constraints are satisfied by a tuple then that tuple matches the query.

max at most these many atomic constraints in the query can be satisfied by a matching tuple.

As an example, consider the query "x=1 AND (y=2 OR z=3)" where x,y,z are field names. The atomic constraints of this query are "x=1", "y=2" and "z=3". And it is easy to argue from the structure of the query that (2, 2, 3) is a triplet for this query.

It is important to observe that the definition does not call for the best possible values of min, exact and max: it is valid to underestimate min and overestimate exact and max. So $(0, \infty, \infty)$ is another triplet for the example query above (in fact it is a triplet for all queries). When calculating these triplets in the server we do want to obtain optimum values (since the efficiency of the index is thereby improved) but we don't have to expend an inordinate amount of time in the calculation: a little sloppiness is tolerable.

The calculation of the triplet for a query is done recursively according to the following theorem (the proof of which is omitted due to space constraints).

Theorem 1 *Let A and B be logical queries with triplets (p, q, r) and (s, t, u) respectively. If A and B have no atomic constraints in common then*

triplet(atomic constraint)	*= (1, 1, 1)*
triplet(other comparisons)	*=* $(0, \infty, \infty)$
triplet(A AND B)	*= (p+s, max(q+u,t+r), r+u)*
triplet(A OR B)	*= (min(p,s), q+t-1, r+u)*
triplet(NOT A)	*=* $(0, \infty, \infty)$

To obtain a list of queries corresponding to a tuple from these index structures the tuple is converted into a list of atomic constraints: if the tuple has values $v_1, v_2, \ldots, v_n$ in fields $f_1, f_2, \ldots, f_n$ the list is "$f_i = v_i$" $i = 1, 2, \ldots, n$.

The index map is probed for each atomic constraint in the list and the corresponding bitmaps (n of them) are added together. The result is a non-negative integer for each query in the system. Let (min, exact, max) be the triplet for the query and let c be the non-negative integer obtained by the bitmap addition for this query. Categorization of all queries into exact, possible and non-matches (with respect to the given tuple) is done as follows:

c $<$ min	$\Rightarrow$ NO match
min $\leq$ c $<$ exact	$\Rightarrow$ POSSIBLE match
exact $\leq$ c	$\Rightarrow$ EXACT match

Exact-match queries don't require further processing. Possible-match queries have to be post-processed against the tuple to check whether they really match. Queries which are an AND combination of distinct atomic constraints (pure AND queries) never require any post-processing. The same applies to pure OR queries. We can also say that queries in conjunctive normal form, in which each disjunction is a OR of mutually exclusive atomic constraints (e.g. "(x = 1 OR x = 2) AND (y = 3 OR y = 4) AND z = 5" will be exact matches as well (provided we special-case the triplet calculation of such queries). Other queries may not always be exact matches: it depends on the optimality of the triplet calculation and the values in the incoming tuple.

5.3. Architecture

The SSE consists of the server and mailer processes which communicate over the network. The server periodically polls the main database to retrieve new subscriptions and new tuples which may match existing subscriptions. It then sends a list of matching (subscription, tuple) pairs to the mailer which performs the task of dynamically generating the notification email and forwarding it to the mail server.

Splitting the functionality into two processes (rather than having a single monolithic process) allows the server and mailer to run on different machines in case of high load. In addition to enabling the administrator to configure the system according to the demands of her site, it also makes for a clean seperation between the subscription (which queries have been matched by the latest tuples) and the notification (how to inform the subscription owner of relevant modifications).

The following subsections give an overview of the various components within these two processes.

5.3.1. RefreshThread and TupleCache: The RefreshThread polls the database on a regular basis for new/updated tuples. All the content tables have a field which stores the timestamp at which the tuple was last modified. There is also an index on this field for all content tables. Therefore, the RefreshThread can do its task very efficiently.

The modified tuples discovered by the RefreshThread are placed in the TupleCache, which is an in-memory store of all recently inserted/updated content tuples. Internally, we use a slotted page structure commonly found in many databases. To avoid expensive system calls for allocating or deallocating memory, the server itself manages a global pool of pages which are requested and returned by the TupleCache.

Inside the TupleCache, tuples are organized on the basis of their content table. That is, tuples from the same content table are stored together. This makes it possible to quickly retrieve (from other parts of the server) all tuples in a particular content table which have been modified within a time window.

5.3.2. QueryThread and QueryIndex: The QueryIndex is the inverse index described in the previous section. It is maintained by the QueryThread which periodically scans the database table for modified subscriptions and makes corresponding changes to the QueryIndex. The QueryThread also maintains one result file per periodicity (instant, hourly, daily and weekly subscriptions). It uses the TupleCache to obtain a list of all content tuples which have been modified since the last iteration. For each tuple in this list, it looks up the QueryIndex to determine all the satisfying subscriptions for that tuple and writes every matching (subscription, tuple) pair to the appropriate result file. At the end of a period, the result file is sent to the mailer for further processing.

5.3.3. ResultThreads and MailThreads: Once the QueryThread has sent a result file to the mailer, it is picked up by a ResultThread. There are 4 ResultThreads, one each for instant/hourly/daily/weekly periods. They parse the result file and generate the notification email which should be sent to the subscription owner. These emails are placed on a queue from where they are forwarded to a mail server by the MailThreads.

5.4. Miscellaneous Issues

5.4.1. Correctness: Any subscription system must satisfy three correctness properties: (1) No subscription should be triggered spuriously (no false positives), (2) every change to content tuples should trigger all matching subscriptions (no false negatives), and (3) no subscription should be triggered more than once for the same change (no duplicates).

The property of no duplicates is ensured by maintaining some state on disk. Part of this state is a timestamp upto which the QueryThread has processed subscriptions and tuples. In the event of a crash, the QueryThread starts processing from this timestamp forward so that tuples are not processed twice. As an additional precaution, the ResultThread maintains persistently for each subscription, the update time of the latest tuple to trigger the subscription. If this thread gets a (subscription, tuple) pair in which the update time of the tuple is less than the time stored with the subscription, the notification is not generated.

The elimination of false positives is a consequence of Theorem 1 and the way we determine satisfying tuples after index lookup (by post-processing partial matches).

To avoid false negatives we have to consider the fact that tuples are inserted into the content tables by different web servers, which may not agree on the current time. To account for this clock skew, the RefreshThread and QueryThread scan for tuples older than the maximum clock skew (rather than for tuples older than the current time). This ensures that no tuples are missed in the processing.

5.4.2. Robustness and Recovery: To recover quickly from crashes the QueryThread keeps most of the index on disk. This allows it to have a fast startup by eliminating the need to read and index all the subscriptions each time. To rectify possible corruption of the on-disk files due to crashes the QueryThread logs the changes to be made to the disk index before actually making them. This write-ahead log allows it to recover a clean version of the index in case of corruption. The QueryThread also keeps a write-ahead log for the four result files.

5.4.3. Scalability: The system needs to scale to lots of subscriptions. Assuming 1 million users and an average of 10 subscriptions per user, the number of subscriptions blows up to 10 million. Even with a conservative estimate of 100 bytes per subscription the amount of data to be indexed is 1 GB.

Handling this much information in a single index structure means manipulating individual bitmaps in excess of 1 MB. We therefore split the index into sub-indexes, each containing upto a few thousand subscriptions. All content tuples now have to be looked up against each sub-index. When a new subscription is encountered, an effort is made to place it in the sub-index which contains subscriptions with which it shares a lot of atomic constraints. In our system, this is easy to do because subscriptions are not arbitrary but drawn from a limited number of pre-defined templates. With this optimization, index splitting is not as much of a performace hit as it would otherwise be.

6. Hierarchy Data Type

In this section, we use the term *hive* (as in "a hive of activity") to denote a website created using the QUIQ application, representing a live, active community that is being continually browsed and posted to. The hierarchy datatypes are used to store the navigational structure of the hive, and one of the challenges is to be able to support changes to this structure with minimal impact on the functioning of the hive.

Using a hierarchical structure to organize large quantities of data is a commonly used technique found in diverse contexts such as the internet, file systems, and libraries for examples. The QUIQ architecture supports such an organization through the Hierarchy data type. A hierarchy is defined to be a set of *nodes* or categories organized as a single, rooted *tree*. A hierarchy *instance* refers to a single hierarchy. Mutiple instances can be defined within a hive and a collection utilizes them by defining possibly several attributes that refer to the instances.

The QUIQ application uses a single hierarchy to organize its collection of questions and its collection of related answers according to a classification of problems. Another hierarchy is used to classify users according to their role in the hive. Finally a combination of the question and role hierarchy is used to determine a user's role with regard to certain categories of questions. As a concrete example, consider a group of *installation* related questions specific to product *xyz*, version *1.2.3*. User *Joe* may be an *expert* for versions 1.2 and higher but a *novice* for earlier versions in which case other users are given immediate feedback regarding the quality of answers given by Joe in various question categories.

When a collection declares a field to use a hierarchy data type, it obtains the following functionality:

- **Multiple Associations:** a record in the collection may be associated with multiple hierarchy nodes from the same instance.
- **Navigation Idioms:** interfaces are automatically generated to support hierarchical navigation through a web page per hierarchical level (i.e. Yahoo [11]) and through pull-down menus.
- **Queries:** are supported for matching a node and descending from a node. In addition, all queries are integrated with the query paradigm discussed in Section 4.3.
- **Modifications:** nodes can be renamed, added, deleted, and moved within the hierarchy. In addition, records may be deleted or moved to other nodes.

The next section describes how the hierarchy data type is stored in order to support the functionality required from queries and restructuring which are discussed in the subsequent sections.

6.1. Storage

The Hierarchy data type is stored in two locations in a hive's database: (1) a separate table is used to store all instance definitions and (2) each record with a declared hierarchy field is set with a hierarchy value. In addition, hierarchy values are tokenized and managed by the Token Index for use in query processing. This section focuses on the database storage of the hierarchy data type.

6.1.1. Hierarchy Definition All definitions per hive of Hierarchy data type instances are stored in their own table where each record represents a hierarchy node. The schema is as follows:

$$< instanceId, parentId, nodeId, prop1, ..., propN >$$

InstanceId differentiates hierarchy instances where nodeId's are unique per instance. The parentId refers to a unique record of an instance that represents the parent node for a node. A parentId of -1 represents the root node of an instance. The N properties are used for exaple, to name a node, or more generally to to flexibly annotating a node.

On startup, the QUIQServer reads all hierarchy definitions into an in-memory data structure that provides more efficient access. Writes are permitted only through the modifications which are described in Section 6.3.

6.1.2. Hierarchy Values The atomic value for a Hierarchy data type value is a single node. A Hierarchy value may be composed of *multiple* nodes from the same hierarchy instance. A value can be associated with any record whose collection schema declares its use of a Hierarchy instance. For such records, a variable length character (varchar) field is used to store the value. The representation of a hierarchy value is a delimited set of atomic node values where each atomic node value is represented by a path and the nodeId of the value's node. The path is a delimited string of nodeId's. Since multiple values may share a common ancestry, compression is achieved by factoring out the greatest common path prefix amongst a record's atomic values.

6.2. Query Processing

The Hierarchy data type supports several useful queries:

1. **Containment ($\subseteq$):** returns all records whose hierarchy value contains an atomic value (excluding the path) equal to the given node.
2. **Subtree ($\leq$):** returns all records whose hierarchy value contains an atomic value equal to the given node or has a path that contains the given node's path.
3. **Proximity ($\sim$):** returns records and scores them by how close they are with respect to the given node. The scoring function and a definition of *close* are flexible.

The Token Index is used to evaluate the above queries. In addition, we gain the flexibility to include the Hierarchy data type in the query paradigm discussed in Section 4.3. As a result, queries over the hierarchy data type have the flexibility to be issued as exact match queries or weighted relative to the importance of matches from the other fields. In addition, query type 3 uses the per-constraint weighting functionality to define scores based on the proximity of records relative to their postition within the hierarchy. The following paragraphs describe what is stored in the Token Index for a Hierarchy value and how the Token Index is used for the above queries.

Tokenization As mentioned in Section 4.3, the Token Index can be used by any data type that provides a *tokenization* method for its values. For a given atomic value, the tokenization method uses two token-fields: *value* and *path*. The value field contains the nodeId of the atomic value whereas the path contains a token for each node in the atomic value's path.

Containment queries are evaluated by probing the Token Index using the *value* token-field. Only those records that contain a value at the specified node will have tokenized such a value token.

Subtree queries are evaluated by probing the *path* token-field. Records in the tree rooted at the subtree will include subtree root nodeId in the path field so will be returned.

Proximity queries to a given node are implemented by issuing a containment queries on the node, its parent, and its siblings. Such queries allow results from hierarchy nodes other than the query node, but *weight* such results lower according to the distance from the query node. Thus, a higly scored result from a sibling node would have to have a very relevant result in some other field to compensate the lower weight attributed to its distance in the hierarchy. The QUIQ architecture does not limit the implementation of hierarchy distance to the above example, however.

In the event that a Token Index is not operational, the QUIQServer is able to process hierarchical constraints using exclusively the database values associated with each record.

6.3. Hierarchy Modifi cations

Changes to a Hierarchy definition are referred to as *modification* events. There are two types of modifcations of interest:

- **Definition:** modify property fields of existing hierarchy node records. These are lightweight operations due to the storage separation of a hierarchy definition and its values associated with records.
- **Structure:** add or delete hierarchy node records or change the parent of a hierarchy node. Additionally, support re-assigning records from one node to another node. Except for the addition of nodes at the leaf level of the hierarchy, the inclusion of ancestry information in the Token Index and database requires propagating the hierarchy modification to all affected record values.

It is assumed that a single user modifies a Hierarchy. Since a Hierarchy modification can effect many parts of an application, the modifications are made to a copy of the Hierarchy. A read-only *preview* mode is provided so that application functionality can be evaluated given the proposed modifications. When the user is satisified with the modifications, they *commit* the changes by scheduling some time in the future for the approproate updates to take place.

Applying hierarchy modifications requires that the system is quiesced. Both definition and structural modifications require this measure since it is assumed that multiple QUIQServers may be accessing the database and we require that all use the same hierarchy definition. Since all QUIQServers cache an in-memory version of hierarchy definitions, when all QUIQServers are restarted, we are guaranteed they will come up with the same view of the hierarchy definitions.

When the system is quiesced, the following steps are run prior to bring the QUIQServers back up:

Definition modifications: The new version of the hierarchy definition replaces the current version and all QUIQServers are brought up.

Structural modifications: These may require downtime proportional to the number of records affected by the modification. The reason is due to redundantly storing a node's ancestry information in the Token Index and the database. However, we expect that structural modifications are relatively rare compared to the number of queries utilizing ancestry information (subtree queries).

The steps taken during a structural modification are best described using an example. Consider inserting a node **B** in the path **A,C** to form the new path **A,B,C**. First, an administrator modifies C's parent to be a new node named **B** whose parent is **A** in the preview version of the hierarchy definition. After the change is commited and the time comes to apply the change, the hive is quiesced to external operations. The modification requires that the new path is inserted in all records of the current subtree under **AB**. Corresponding changes are required to the Token Index. In order to efficiently find the required records, the Token Index is used to evaluate the subtree query and retrieve all results. Then, the appropriate changes are propagated to all retrieved records in the Token Index and database. As with definition modifications, the preview version takes the place of the current version and the QUIQServers are restarted.

7. Data Warehouse

The data warehouse is used for data mining and report generation, and has a standard design. Keeping the report data separate from the *online* data is important to prevent expensive reporting queries from causing the online transaction database from slowing down. The data in the data warehouse is also in a denormalized format that facilitates analysis, and additional aggregation tables are included in order to reduce the number of group by queries and allow different reporting tools to easily access the data. The data warehouse is typically used in conjunction with a reporting engine.

The tables in the warehouse are populated through an hourly data pull process. Every hour, updated records are transferred from the transactional database tables to the warehouse. Typically, only a small number of records are pulled every hour as the modifed data is easily identified by the update time. In addition to the transactional tables, data from the webserver log, which is instrumented to provide context about each user-click, is also consolidated into a warehouse table. Along with the basic reporting tables in the warehouse, there is a metadata table which stores the mappings between the transactional tables and the basic reporting tables, and the timestamp in the transactional table of when the data was last pulled. This is used to determine which data to pull. After all the basic reporting tables are built, the queries to build the aggregate queries are run.

8. Related Work

The concept of mass collaboration is being harnessed to solve complex problems other than customer support. For example, the Cooperative Bug Isolation Project at UC Berkeley [15, 16] tries to find bugs in software by aggregating traces from many users. Similarly, mass collaboration is being proposed for data integration as in [18].

The multi-tiered architecture is similar to standard multi-tiered applications that are backed by a database. In the case of the QUIQ architecture, a custom application server is used in order to combine and manage the heterogenous data services needed by the QUIQ application. The discussion in [21] provides more details and examples of multi-tiered architectures.

The functionality offered by the permissioning system is flexible fine-grained access control (record attribute value)for higher layers of QUIQ application code. The finest granularity supported by commercial database systems is row-level access control [9]. Fine grained access control is also supported by the SeaView system [17] but its implementation details and performance characteristics are unknown. The implementation of QUIQ permissions is through query modification. Given a query and user context, the query is modified according to the user's capabilities to provide a data-driven form of access control: a combination of data values and capability values determine what subset of data can be manipulated by the user. Query modification as a means for implementing access control was first proposed in [24]. Content management systems such as [10] also enforce access control through an application server layer.

The Token Index is central to the integration of text and tables in the QUIQ system. Commercial RDBMSs have been extended to allow keyword searches over textual attributes of the tuples in the database. Similarly, some text indexing engines allow some non-text attributes to be associated with them. However, non-text attributes in either case simply filter the results whereas the Token Index integrates them into computing relevance scores.

With regard to implementation, deferring updates has been extensively studied in both the context of text and non-text attribute indexing. The points of comparison include:

Propagation Style: Is the index modified by **re-writing** a portion of the index or by reserving free-space in order to apply the update *in-place*?

Granularity of Propagation: What is the unit that written: a posting list entry, a posting list, a group of posting lists, etc.?

Propagation Frequency: When is a change propagated?

The Token Index propagation style is re-write, its granularity is a partition which is a collection of posting lists, and the propagation frequency is periodic such that the whole index is rewritten in 24 hours. In contrast, the approach taken in [3] uses a posting list as its granularity and its propagation frequency is determined by the amount of available memory. The publicly available search engine framework Lucene in [6, 5] uses a granularity and propagtion frequency that is based on available memory. When memory fills, it is flushed into a new file which is a propagation unit. After a number of these accumulate, they are merged into a single propagation unit, thus resulting in propagation units whose sizes are geometrically related. In contrast, the Token Index mainains a constant number of propagation units over time but is less flexible if memory is exceeded. The text retrieval system presented in [4] similarly propagates periodically and uses a fixed-space propagation unit.

In contrast to the Token Index and above systems' use of re-write, the system described in [2] uses a mixed approach due to its use of relying on variable sized allocation units. If an update to a posting list can fit in an existing, allocated unit, the change is propagated in-place. Otherwise, the entire posting list is re-written to a new, possibly larger sized unit. The study in [25] considers various alternatives for how to propagate updates. The results highlight the tradeoff between in-place and rewrite strategies with respect to update versus query performance. In-place results in greater fragmentation which hurts queries but results in less work for udaptes. Rewrite on the other hand causes less fragmentation at the cost of reading and writing more data.

All of the systems above assume a single field and assume that a field field identifier is transient. In contrast, the Token Index must keep track of multiple fields per record and it must maintain the correspondance between its field values and record identifiers present in the external DBMS. This functionality is similar to that required by commercial RDBMSs.

In the context of indexing non-text fields, several indexing schemes have been developed in order to find the right balance between update and query throughput. The work in [23] argues for leaving a small area on disk for deferring changes so that the larger, existing dataset can be better organized on disk. A technique for answering queries over the union of the structures is also proposed. Data warehouses also need to accomodate changes to a structure that is highly optimized for queries. The work in [12] proposes that the changes are similarly deferred and merged into the main stcuture using a multi-level merge algorithm that depends on either hashing or sorting. Another multi-level approach is proposed in [20, 19] where the components of the merge and merge algorithm are carefully designed in order to maximize sequential disk usage.

The SSE's functionality is closest to *continuous query* systems. An online survey of the research pertaining to this field can be found at [7]. In addition, the SQL Server commercial RDBMS [8] supports similar functionality in its *Notification Server* component.

9. Conclusion

We presented an overview of the QUIQ mass collaboration architecture, which was designed and developed in 1999 and 2000, and first deployed in the Ask Jeeves AnswerPoint service in 2000. QUIQ was acquired by Kanisa in 2003, but the application continues to be in use (e.g., the Compaq service community). The architecture reflects the challenges in buidling an application that requires tight integration of text and database systems, and this is a direction that database vendors are currently working hard to provide improved support. Recent developments such as Microsoft's Notification Server extension to SQL Server are also steps in the right direction; the lack of such a capability in 1999 led us to develop the SSE. It was disappointing that SQL's authorization mechanisms were so inadequate that we had to develop, in essence, a stand-alone role-based, fine-grained access control mechanism in our application. With an increasing emphasis on privacy and secure access, this is another area to extend support within standard SQL systems. Finally, we believe that the paradigm of mass collaboration will find increasing use, and the QUIQ application made a compelling case for it in an intensely measured and analyzed domain, corporate customer support.

10. Acknowledgments

A number of people made major contributions to the design and development of the QUIQ system, including Luke Blanshard, Prasad Deshpande, Harvey Goodman, Jim Kupsch, Doug Leavitt, Beth Martinson, Paul Parter, Kartik Ramakrishnan, Rajesh Raman, Amit Shukla, Josh Solomon, and Markus Zirn.

References

[1] S. Bhattacharya, C. Mohan, K. W. Brannon, I. Narang, H.-I. Hsiao, and M. Subramanian. Coordinating backup/recovery and data consistency between database and fi le systems. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 500–511. ACM Press, 2002.

[2] E. Brown, J. Callan, and W. Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, pages 192 – 202, Santiago, Chille, September 1994.

[3] T.-C. Chiueh and L. Huang. Effi cient real-time index updates in text retrieval systems.

[4] C. Clarke, G. Cormack, and F. Burkowski. Fast inverted indexes with on-line update, 1994.

[5] D. Cutting. jakarta.apache.org/lucene.

[6] D. Cutting and J. Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of the 13th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–411, 1990.

[7] http://www.cs.brown.edu/research/aurora/related.html.

[8] http://www.microsoft.com/sql/default.asp.

[9] http://www.oracle.com/solutions/security/Privacy9i.pdf.

[10] http://www.vignette.com.

[11] http://www.yahoo.com.

[12] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental organization for data recording and warehousing. In *The VLDB Journal*, pages 16–25, 1997.

[13] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The quiq engine: A hybrid ir-db system. In *ICDE*, 2002.

[14] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The quiq engine: A hybrid ir-db system. Technical Report TR-1449, University of Wisconsin-Madison, 2002.

[15] B. Liblit. http://www.cs.berkeley.edu/ liblit/sampler.

[16] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Public deployment of cooperative bug isolation. In *2nd International Conference on Software Engineering (ICSE) Workshop on Remote Analysis and Measurement of Software Systems*, 2004.

[17] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. The seaview security model. *IEEE Trans. Softw. Eng.*, 16(6):593–607, 1990.

[18] R. McCann, A. Doan, V. Varadaran, A. Kramnik, and C. Zhai:. Building data integration systems: A mass collaboration approach. In *WebDB*, 2003.

[19] P. Muth, P. E. O'Neil, A. Pick, and G. Weikum. The LHAM log-structured history data access method. *VLDB Journal: Very Large Data Bases*, 8(3–4):199–221, 2000.

[20] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree), 1996.

[21] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. WCB McGraw-Hill, 2004.

[22] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. In *Information Processing and Management*, volume 24, pages 513–523, 1988.

[23] D. Severance and G. Lohman. Differential fi les: Their application to the maintenance of large databases. *ACM TODS*, 1(3):256–267, September 1976.

[24] M. Stonebraker and E. Wong. Access control in a relational data base management system by query modifi cation. In *Proceedings of the 1974 annual conference*, pages 180–186. ACM Press, 1974.

[25] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in distributed text document retrieval systems. In *PDIS*, pages 8–17, 1993.