Bulletin of the Technical Committee on

Data Engineering

June 2004 Vol. 27 No. 2

IEEE Computer Society

Letters

Special Issue on Database Caching Strategies for Web Content

Conference and Journal Notices

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

There are two Data Engineering Bulletin web sites: http://www.research.microsoft.com/research/db/debull and http://sites.computer.org/debull/.
The TC on Data Engineering web page is http://www.ccs.neu.edu/groups/IEEE/tcde/index.html.

Letter from the Editor-in-Chief

Bulletin Editors

I am very proud that issue after issue of the Data Engineering Bulletin exemplifies such consistently high quality. This pride, to be fully honest, is one level indirect. It is the issue editors who select the topic, solicit the articles, and ride herd on the authors so as to get the Bulletin published. So the great bulk of the credit over the years is rightly assigned to our associate editors. My pride is in having chosen the editors so successfully.

Associate editors serve for two years. We are at a junture now in which old editors are leaving and new editors are arriving. So it is time now to thank Umesh Dayal, Johannes Gehrke, Christian Jensen, and Renée Miller for the outstanding jobs that they have done. It has been a pleasure working with them to publish the fine issues of the last two years.

It is also time to welcome new editors. There are four new editors, whose names now appear on the inside front cover: Gustavo Alonso of ETH, Minos Garofalakis of Bell Labs, Meral Ozsoyoglu of Case Western Reserve, and Jignesh Patel of Michigan. This new crop of editors continues my ”lucky streak” of succeeding in attracting outstanding editors for the Bulletin. This is the most important part of my job as editor-in-chief. The great new issues from these editors begin with the current issue, which is described below.

The Current Issue

Performance of our systems has always been important. But increasingly, it is scalability that is the ”performance” goal for systems. Implementors are willing to trade some system efficiency for an ability to scale the system to very high performance via adding additional resources. Here is where caching and replication come into play.

Maintaining data replicas consumes system execution cycles. However, replication of data enables system performance to scale by the addition of more processors and disks. The current issue explores replication and caching strategies. There are several important considerations in this. One isse is the extent to which applications need to know where data is stored or replicated. This is transparency. Another issue is one of consistency. If one accesses data stored across multiple replicas, how can one specify what is wanted in the way of consistency? Is transactional consistency needed or some other measure. And how does a system provide this consistency? Yet another issue is currency. How are replicas maintained, and how can they guarantee something with respect to how recent the information is in a replica? And how is it decided where the replicas should go?

The current issue explores the above issues and others. It brings together a mix of papers from industrial, academic and government researchers. I have long thought that having industrial participation is an important part of the Bulletin mission, even when the industrial people are *researchers*. And this is surely successful in the current issue. I am sure that readers will be well rewarded in reading about work which is actively being pursued across such a wide spectrum of institutions. Many thanks to Gustavo Alonso for bringing us this highly useful and interesting collection of articles on a very timely subject.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

Data replication has always been an important topic. No matter in what context, local access to the data is faster than accessing a remote server. The proliferation of networks has only made this basic principle more important when designing large scale, distributed information systems. Data replication has nevertheless taken many different forms depending on the type of application: caching, lazy replication, eager replication, etc. As the contributions in this special issue demonstrate, the need to provide faster access to dynamic content is blurring these differences in implementation. On the one hand, database caching is becoming the basis for replicating data in remote locations at the edge of the network to provide fast access to dynamic content. On the other hand, data replication techniques are being used to extend the reach of databases beyond the local are network to make them more flexible and applicable in a wider range of contexts. Given the explosive growth of dynamic content and the need for novel solutions in service oriented architectures and commodity computing, data replication will certainly remain an important research topic for years to come.

This issue contains a collection of five papers that address the problem of data replication and caching from a variety of points of view. The first paper, by Theo Härder and Andreas Bümann of the University of Kaiserslautern, addresses the basic issue of populating a cache with the data necessary to answer queries locally. The paper uses the notions of *value completeness*, *value range completeness*, *domain completeness*, and *predicate completeness* to define several fundamental aspects of the database caching problem. The second paper, by Christoph Bornhövd, Mehmet Altinel, C. Mohan, Hamid Pirahesh, and Berthold Reinwald, describes *DBCache*, an extension to DB2 that supports database caching at both the edge of content delivery networks and at the middle-tier of Web site infrastructures. The novelty behind DBCache is that it blurs the notion of data replication and data caching thereby allowing to extend the concept of database caching well beyond conventional applications. The third paper, by Emmanuel Cecchet, presents *C-JDBC*, an open source system that implements the Redundant Array of Inexpensive Databases (RAIDb) concept at the database level. C-JDBC primarily targets J2EE application servers requiring database clusters build with commodity hardware and is therefore an important complement to the idea presented in the first two papers. The fourth paper, by Christian Plattner and Gustavo Alonso, presents *Ganymed*, a flexible data replication solution that exploits snapshot isolation to provide scalability without loss in consistency. Ganymed can be used to implement a database cluster –like C-JDBC– or for caching on the edges of a network –like DBache. The fifth paper, by Per- Åke Larson, Jonathan Goldstein, Hongfei Guo, and Jingren Zhou, describes *MTCache* a mid-tier caching solution for SQL Server. The goal of MTCache is to transparently off-load part of the query workload from a backend server to front-end servers, thereby improving system throughput and scalability but without requiring application changes.

Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology (ETH Zurich) Zurich
Switzerland

Query Processing in Constraint-Based Database Caches

Theo Härder Andreas Bühmann
Department of Computer Science, University of Kaiserslautern, Germany
{haerder, buehmann}@informatik.uni-kl.de

Abstract

Database caching uses full-fledged DBMSs as caches to adaptively maintain sets of records from a remote DB and to evaluate queries on them, whereas Web caching keeps single Web objects ready somewhere in caches in the user-to-server path. Using DB caching, we are able to perform declarative and set-oriented query processing near the application, while data storage and consistency maintenance is remote. We explore which query types can be supported by DBMS-controlled caches whose contents are constructed using parameterized cache constraints. Schemes on single cache tables or on cache groups correctly perform local evaluation of query predicates. In practical applications, only safe schemes guaranteeing recursion-free load operations are acceptable. Finally, we comment on future application scenarios and research problems including empirical performance evaluation of DB caching schemes.

1 Introduction

Database caching tries to accelerate query processing by using full-fledged DBMSs to cache data in wide-area networks close to the applications. The original data repository is a backend database (BE-DB), which maintains the transaction-consistent DB state, and up to n frontend databases (FE-DBs) may participate in this kind of "distributed" data processing. For example, server-selection algorithms enable (Web) clients to determine one of the replicated servers that is "close" to them, which minimizes the response time of the (Web) service. This optimization is amplified if the invoked server can provide the requested data—frequently indicating geographical contexts. If a query predicate can be answered by the cache contents, it is evaluated locally and the query result is returned to the user. When only the answer to a partial predicate is locally available, the remaining part of the predicate is sent to the BE-DB. Obviously, this kind of federated query processing, where the final query result is put together by the FE-DB, can be performed in parallel and can save substantial communication cost.

An FE-DB supports declarative and set-oriented query processing (e. g., specified by SQL) and, therefore, keeps sets of related records in its DB cache which must satisfy some kind of *completeness condition* w. r. t. the predicate evaluated to ensure that the query execution semantics is equivalent to the one provided by the BE-DB. Updates to the database are handled—in the simplest scenario—by the BE-DB which propagates them to the affected FE-DBs after the related transaction commit. We assume that an FE-DB modifies its state of the cache within a time interval δ after the update, but do not discuss all the intricacies of DB cache maintenance.

So far, all approaches to DB caching were primarily based on materialized views and their variants [2, 4]. A materialized view consists of a single table whose columns correspond to the set of output attributes $O_V = \{O_1, \ldots, O_n\}$ and whose contents are the query result V of the related view-defining query Q_V with predicate P.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Materialized views can be loaded into the DB cache in advance or can be made available on demand, for example, when a given query is processed the nth time ($n \geq 1$), exhibiting some kind of built-in locality and adaptability mechanism. When they are used for DB caching, essentially independent tables, each representing a query result V_i of Q_{V_i}, are separately cached in the FE-DB. In general, query processing for an actual Q_A is limited to a single cache table. The result of Q_A is contained in V_i, if P_A is logically implied by P_i (subsumption) and if O_A is contained in O_{V_i}. Only in special cases, a union of cached query results, e. g., $V_1 \cup V_2 \cup \ldots \cup V_n$, can be exploited. DBProxy [2] has proposed some optimizations at the storage level. To reduce the number of cache tables, DBProxy tries to store query results V_i with strongly overlapping output attributes in common tables.

2 Concept of constraint-based database caching (CBDC)

CBDC promises a new quality for the placement of data close to their application. The key idea is to accomplish for some given types of query predicates P the so-called *predicate completeness* in the cache such that all queries eligible for P can be evaluated correctly. All records (of various types) in the BE-DB which are needed to evaluate predicate P are called the *predicate extension* of P. Because predicates form an intrinsic part of a data model, the various kinds of eligible predicate extensions are data-model dependent, that is, they always support only specific operations of a data model under consideration.

Suitable cache constraints for these predicates have to be specified for the cache. They enable cache loading in a constructive way and guarantee, when satisfied, the presence of their predicate extensions in the cache. The technique does not rely on the specification of static predicates: The constraints are parameterized making this specification adaptive; it is completed when the parameters are instantiated by specific values. An "instantiated constraint" then corresponds to a predicate and, when the constraint is satisfied—i. e., all related records have been loaded—it delivers correct answers to eligible queries. Note, the union of all existing predicate extensions flexibly allows the evaluation of their predicates, i. e., $P_1 \cup P_2 \cup \ldots \cup P_n$ or $P_1 \cap P_2 \cap \ldots \cap P_n$ or subsets/combinations thereof, in the cache.

There are no or only simple decidability problems whether predicates can be evaluated. Only a simple probe query is required at run time to determine the availability of eligible predicate extensions. Furthermore, because all columns of the corresponding BE tables are kept, all *project operations* possible in the BE-DB can also be performed in the cache. Other operations like *selection and join* depend on specific *cache constraints*. Since full DB functionality is available, the results of these queries can further be *refined* by selection predicates such as Like, Null, etc. as well as processing options like Group-by, Having (potentially restricted), or Order-by.

A cache contains a collection of cache tables which can be isolated or related to each other in some way. For simplicity, the names of tables and columns are identical in the cache and in the BE-DB. Considering a cache table S, we denote by S_{B} its corresponding BE table, by $S.c$ a column c of S. Note, a cache usually contains only subsets of records pertaining to a small fraction of BE tables. Its primary task is to support local processing of queries that typically contain simple projection (P) and selection (S) operations or such having up to 3 or 4 joins (J) [1]. Hence, we expect the number of cache tables—featuring a high degree of reference locality—to be in the order of 10 or less, even if the BE-DB consists of hundreds of tables.

3 Supporting PS queries

Let us begin with single cache tables. If we want to be able to evaluate a given predicate in the cache, we must keep a collection of records in the cache tables such that the completeness condition for the predicate is satisfied. For simple equality predicates like $S.c = v$ this completeness condition takes the shape of *value completeness*.

Definition 1 (Value completeness, VC): A value v is said to be value complete in a column $S.c$ if and only if all records of $\sigma_{c=v} S_{\mathrm{B}}$ are in S.

If we know that a value v is value complete in a column $S.c$, we can correctly evaluate $S.c = v$, because all rows from table S_B carrying that value are in the cache. But how do we know that v is value complete? This is easy if we maintain *domain completeness* of specific table columns.

Definition 2 (Domain completeness, DC): A column $S.c$ is said to be domain complete (DC) if and only if all values v in $S.c$ are value complete.

Given a DC column $S.c$, if a probe query confirms that value v is in $S.c$ (a single record suffices), we can be sure that v is value complete and thus evaluate $S.c = v$ in the cache. Note that unique (U) columns of a cache table (defined by SQL constraints "unique" or "primary key" (PK) in the BE-DB schema) are DC per se (*implicit* domain completeness). Non-unique (NU) columns in contrast need extra enforcement of DC.

3.1 Equality predicates

If a DC column is referenced by a predicate $S.c = v$ and v is found in $S.c$, then table S is eligible for a PS query containing this equality predicate. To explicitly specify a column to be domain complete, we introduce a first cache constraint called *cache key column*, cache key for short[1], which can always be used as an entry point for the evaluation of equality predicates. But in addition, a cache key serves as a *filling point* for a table S. In contrast to [1], we define cache keys in a more subtle way. Because low-selectivity values even in a high-selectivity column defined as cache key column can cause filling actions involving huge sets of records never used later, filling control by a recommendation list R or stop-word list L is mandatory. Whenever a query references a particular cache key value v that is not in the cache, query evaluation of this predicate has to be performed by the BE-DB. However, as a consequence of this cache miss attributed to a cache key reference, the cache manager satisfies value completeness for v—if it is contained in R (or not contained in L)—by fetching all required records from the backend and loading them into the table S (thus keeping the cache key column domain complete[2]).

Definition 3 (Cache key column): A cache key column $S.k$ is always kept domain complete. Only values in $R \subseteq \pi_k S_B$ initiate cache loading when they are referenced by user queries.

Hence, a reference to a cache key value x—or, more general, to a cache constraint—in a query serves as something like an indicator that, in the immediate future, locality of reference is expected on the predicate extension determined by x. Cache key values therefore carry information about the future workload and sensitively influence caching performance. Therefore, usage statistics and history information collected for such references should be employed to select cache keys and to build recommendation lists.

3.2 Range predicates

To answer range queries on a single column, we need a new type of cache constraint called cache value range which makes all values of a specified range value complete. Although possible for any domain with ordered values, here we restrict our considerations to domains of type integer or string. Hence, a value range carries two parameters l and u ($-\infty \leq l \leq u \leq +\infty$).

A value range defined with closed boundaries serves as another type of cache constraint. In a query, a range predicate can take various forms using the relationships $\Theta \in \{<, =, >, \leq, \neq, \geq\}$. An actual range predicate r_A can be easily mapped to a cache value range, e. g., $x > l$ to $l \leq x < +\infty$. When loading range predicate extensions, value ranges r_i already existing in the cache have to be taken into account. In the simplest case, the cache could be populated by all records belonging to the (partial) range which leads to a cache miss when a range query with $S.c = r = (l \leq x \leq u)$ is evaluated. Cache loading makes table S range complete for $S.c = r$ such that subsequent queries with range predicates contained in r can be correctly answered in the cache.

[1] We assume single-column cache keys. An extension of our statements to multi-column cache keys, however, is straightforward.

[2] VC for each cache key value keeps probing simple; cache update operations may require a revised definition and complex probing.

Definition 4 (Value range completeness, RC): A value range $r = (l \leq x \leq u)$ is called range complete in a column $S.c$ if and only if all records of $\sigma_{c \geq l \wedge c \leq u} S_{\mathrm{B}}$ are in S (making all individual values in $S.c$ value complete).

To be aware of the value ranges r_i present in the cache, the cache manager keeps an ordered list of the r_i. As soon as two adjacent r_i merge, they are melted to a single range. Hence, queries with range predicates r_A contained in an r_i present ($r_A \subseteq r_i$) can be locally evaluated, whereas overlapping r_A cause the cache to be populated by records making the missing values RC. Note, the selectivity and potential locality of key ranges have to be strictly controlled to prevent "performance surprises". This is especially true for open ranges (l or u is ∞) or $\Theta = \neq$. Again, cache filling should be refined by a recommendation list R (or stop-word list L).

Definition 5 (Cache value range, CVR): A column $S.k$ is domain complete and each value range r_i is always kept range complete. Only values of r_i in $R \subseteq \pi_k S_{\mathrm{B}}$ initiate cache loading when they are referenced by queries.

3.3 Other types of predicates

SQL allows some more types of predicates on single tables. However, although possible, it is not reasonable to strive for keeping their predicate extensions in the cache. For predicates which need large portions of or even the entire table for their evaluation, it is more reasonable to process the related query in the BE-DB and to provide a materialized view in the cache. This is usually true for all aggregate queries (MAX, MIN, SUM, user-defined aggregate functions, etc.) or queries containing *like* predicates. Of course, depending on the specific parameters and references, predicates such as Exists, All, etc. or those requiring subqueries are "bad" for the use of CBDC, because their predicate extensions may be too large and may not exhibit enough reference locality.

However, if queries contain a constraint-determining (part of a) predicate such as an equality or range predicate P, these "bad" types of predicates, when limited to predicate extensions of P, can be applied in the cache thereby allowing perfectly local query processing.

4 Support of PSJ queries

A powerful extension of cache use is to enable equi-joins to be processed in the cache and to combine them with PS predicates, thereby achieving PSJ queries. For this purpose, we introduce *referential cache constraints* (RCCs), which guarantee the correctness of equi-joins between cache tables. Such RCCs are specified between two columns of cache tables S and T, which need not be different, not even the columns themselves.

Definition 6 (Referential cache constraint, RCC): An RCC $S.a \rightarrow T.b$ between a source column $S.a$ and a target column $T.b$ is satisfied if and only if all values v in $S.a$ are value complete in $T.b$.

RCC $S.a \rightarrow T.b$ ensures that, whenever we find a record s in S, all join partners of s with respect to $S.a = T.b$ are in T. Note, the RCC alone does not allow us to correctly perform this join in the cache: Many rows of S_{B} that have join partners in T_{B} may be missing from S. But using an equality predicate on a DC column $S.c$ as an "anchor", we can restrict this join to records that exist in the cache: The RCC $S.a \rightarrow T.b$ expands the predicate extension of $S.c = x$ to the predicate extension of $(S.c = x$ and $S.a = T.b)$. In this way, DC columns serve as *entry points* for queries. In a similar way, CVRs can be combined with RCCs to construct predicate extensions (for range and join predicates) to be locally evaluated. For the implementation of this idea, we refer to a particular approach called *cache groups*.

Definition 7 (Cache group): It is a collection of cache tables linked by a number of RCCs. A distinguished cache table is called the root table R of the cache group and holds at least one cache key or cache value range. The remaining cache tables are called member tables M_i and must be reachable from R via the (paths of) RCCs.

Figure 1: Cache groups COL and COP for order processing

Cache constraints—cache keys as filling points defined on R and RCCs specified between R and M_i—determine the records of the corresponding BE tables that have to be kept in the cache. Depending on the types of the source and target columns on which an RCC is defined, we classify the RCCs as U $\rightarrow$ U or U $\rightarrow$ NU (member constraint, MC), NU $\rightarrow$ U (owner constraint, OC), and NU $\rightarrow$ NU (cross constraint, XC).

4.1 TimesTen cache groups

The TimesTen (TT) team originally proposed the notion of a cache group [5] that consists of a root table and a number of member tables connected via member constraints (corresponding to PK/FK relationships in the BE-DB). A TT cache instance (CI) is a collection of related records that are uniquely identifiable via a CI key. The root table carries a single identifier column (U) whose values represent CI keys. Because all records of CIs must be uniquely identifiable, they form non-overlapping tree structures (or simple disjoint DAGs) where the records embody the nodes and the edges are represented by PK/FK value-based relationships.

Note, there is no equivalence to our notion of cache keys (possibly NU), because cache loading is not based on reference to specific values (parameterized loading). In contrast, it is controlled by the application (which gives something like prefetching directives or hints) where various options are supported for the loading of CIs ("all at once", "by id", "by where clause"). There is no notion of domain completeness, of cache-controlled correctness, or of the completeness of predicate extensions. Figure 1 illustrates two cache groups with tables C,O,L and P where $C.a$, $O.d$, and $P.e$ are U columns and $O.b$, $O.c$, and $L.e$ are NU columns. In a common real-world situation, C,O,L and P could correspond to BE-DB tables Customer, Order, OrderLine and Product. COL is a TT cache group and is formed by $C.a \rightarrow O.b$ and $O.d \rightarrow P.e$ where both RCCs would typically characterize PK/FK relationships used for join processing. For example, if Customer $(C.a = 4711)$ as CI key is in the cache, its CI represents a tree structure used to locate all of his Orderline records as leaves.

4.2 Cache groups

Cache groups fully adhere to our definitions of cache keys and RCCs. They are introduced by the DBCache project [1] and extend the TT cache groups by enabling the use of RCCs of types OC and XC—in addition to MC—and by explicit specification of cache keys which make them parameterizable and (weakly) adaptive. Despite similarities, MCs and OCs are not identical to the PK/FK relationships in the BE tables: Those can be used for join processing symmetrically, RCCs only in the specified direction. XCs have no counterparts at all in the BE-DB. Because a high fraction of all SQL join queries refers exclusively to PK/FK relationships—they represent real-world relationships captured by the DB design—, almost all RCCs are expected to be of type MC or OC; accordingly XCs and multiple RCCs ending on a NU column seem to be rare.

Query processing power and flexibility of cache groups are enhanced by the fact that specific columns are made implicitly domain complete via our RCC mechanism (for details, see [3]).

Definition 8 (Induced domain completeness, IDC): A cache table column is induced domain complete, if it is the only column of a cache table filled via one or more RCCs or via a cache key definition.

If a probing operation on some explicitly specified or implicitly enforced domain-complete column $T.c$ identifies value x, we can use $T.c$ as an entry point for evaluating $T.c = x$. Now, any enhancement of this predicate with equi-join predicates is allowed if these predicates correspond to RCCs reachable from table T.

Cache group COP in Figure 1 is formed by $C.a \rightarrow O.b$ and $O.c \rightarrow P.e$, and carries $C.t$ as a cache key. In COP, CIs form DAGs, and a single cache key value may populate COP with a set of such CIs. If we find 'gold' in $C.t$, then the predicate ($C.t =$ 'gold' and $C.a = O.b$ and $O.c = P.e$) can be processed in the cache correctly. Because the predicate extension (with all columns of all cache tables) is completely accessible, any column may be specified for output. Additional RCCs, for example, $C.t \rightarrow O.b$ or $O.c \rightarrow C.n$ are conceivable (but only useful, if their values are join compatible); such RCCs, however, have no counterparts in the BE-DB schema and, when used for a cross join of C and O, their contributions to the query semantics remain in the user's responsibility. In both cache groups, $O.b$ has IDC and $O.d$ is domain complete by definition. Hence, they can be used as entry points for equality predicates. If, for example, $O.d = x$ is found in the cache, predicate ($O.d = x$ and $O.c = P.e$) can be correctly evaluated. Of course, a correct predicate can be refined by "and-ing" additional selection terms (referring to cache tables) to it; e. g., ($C.t =$ 'gold' and $C.n$ like 'Smi%' and $O.e > 42$ and $\ldots$).

4.3 Enhancement for outer joins

Join processing in cache groups can be enhanced to support outer join operations. An outer join along an RCC R can be evaluated correctly, if anchored at the starting column of R (via some predicate P)—just like ordinary equi-joins. Assume a left outer join to be processed in cache group COP: (C Left Join O Where $C.t$ = 'gold'). RCC $C.a \rightarrow O.b$ guarantees that all (equi-)join partners of C tuples (in the cache) are in O; for the left outer join, those C tuples that do not have join partners receive artificial ones in the cache just like in the BE-DB. In a similar way, full or right outer joins can be achieved if they are supported by appropriate cache constraints.

To summarize our discussion so far, it has revealed that RCCs alone do not allow us to correctly perform joins in the cache. But using an equality or value range predicate on a DC column $S.c$ as an "anchor", we can restrict joins to records that exist in the cache: Hence, an RCC $S.a \rightarrow T.b$ expands the predicate extension of $S.c = x$ to the predicate extension of ($S.c = x$ and $S.a = T.b$). In this way, DC columns serve as entry points for queries for which predicate completeness can be assured.

Definition 9 (Predicate completeness, PC): A collection of tables is said to be predicate complete with respect to predicate P if it contains all records needed to evaluate P, that is, its predicate extension.

5 Other types of queries

So far, we have outlined the most important cache constraints and their resulting predicate extensions to enable correct query evaluation in the cache. To indicate that CBDC can be streched even further, we briefly remark that queries exhibiting other predicate types—less important from a practical view—can be locally processed, too.

5.1 Processing of set operations

If record sets S and T are union compatible—having the same number of attributes mapped pairwise to the same domains—and can be derived from predicate extensions in the cache, then the usual relational set operations can be applied to them, that is, Union ($S \cup T$), Intersection ($S \cap T$), and Difference ($S \setminus T$). Typically, S and T themselves are the results of PS or PSJ queries supported by cache constraints introduced so far.

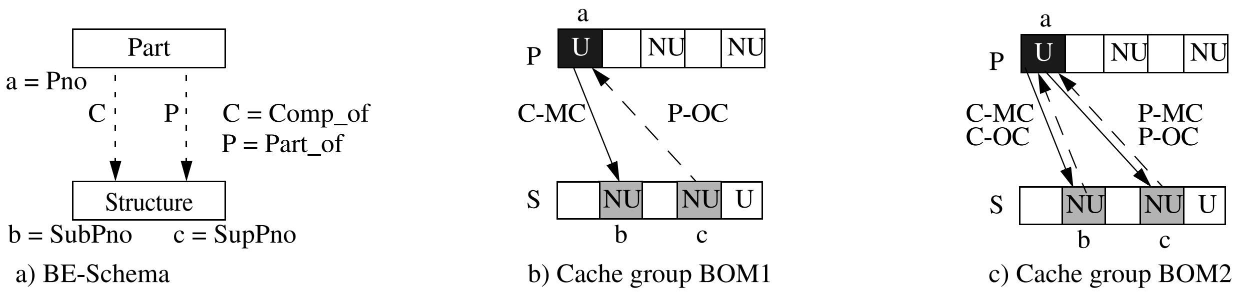

Figure 2: Cache groups for bill-of-material processing (BOM)

5.2 Evaluation of recursive queries

Cache constraints allow us to specify that the cache be recursively populated. Hence, the corresponding predicate extensions embody recursively applied relationships. Figure 2 illustrates some simple examples. Figure 2a represents the relational BE schema of a bill-of-material database. Pno is the primary key (PK) of table Part and SubPno and SupPno are foreign keys (FK) of table Structure where both PK/FK pairs form the relationships Comp_of and Part_of. These relationships are value based and symmetric. In contrast, the corresponding RCCs, although also value based, are directed. Figure 2b–c define two cache groups exhibiting simple and double recursion when the cache groups are populated (via reference of a cache key value of $P.a$). As an instructive exercise, the reader may provide a BOM example for the BE schema in Figure 2a and show the CI of BOM1 for top-level part $P.a = 1$. Then, he may provoke loading of BOM2 for any, even elementary, part $P.a = x$. To be brief, as soon as the predicate extension for $P.a = 1$ is loaded, BOM1 may be used to correctly deliver all component and elementary parts of product $P.a = 1$ (using an SQL query 'With Recursive ... Union All ...').

6 Safeness issues and cache modification

In practical applications, it is mandatory to prevent uncontrollable and excessive cache population as a consequence of recursive dependencies. Although loading can be performed asynchronously to the transaction observing the cache miss and, therefore, a burden on its own response time can be avoided, uncontrolled loading is undesirable for the following reason: It may influence the transaction throughput in heavy workload situations, because substantial extra work to be hardly estimated may be required by the FE-DB and BE-DB servers.

For this reason, only *safe* cache groups exhibiting recursive-free loading may be acceptable. Therefore, some restrictions should be applied to cache group design [3]. The most important one is related to NU-DC columns. When two or more NU-DC columns of a cache table must be maintained, then these columns may receive new values in a recursive way. Hence, in the same cache table, two or more NU columns that are DC are not allowed. Another restriction applies to heterogeneous RCC cycles in cache groups where in some table two columns are involved in the cycle.

In this document, we have excluded all aspects of cache maintenance. How difficult is it to cope with the units of loading and unloading? Let us call such a unit cache instance (CI). Depending on their complexity, CIs may exhibit good, bad, or even ugly maintenance properties. The good CIs are disjoint from each other and the RCC relationships between the contained records form trees (Figure 1 left). The bad CIs form DAGs and weakly overlap with each other (Figure 1 right). Hence when loading a new CI, one must beware of duplicates. Accordingly, shared records must be removed only together with their last sharing CI. To maintain cache groups with strongly overlapping CIs can be characterized as ugly, because duplicate recognition and management of shared records may dominate the work of the cache manager. Again, unsafe cache groups need not be considered at all. In general, they may feature unmanageable maintenance due to their recursive population behavior.

Other interesting research problems occur if we apply different update models to DB caching. Instead of processing all (transactional) updates in the BE-DB first, they could be performed in the FE-DB (under ACID protection) or even jointly in FE- and BE-DB under a 2PC protocol. Such update models may lead to futuristic considerations where the conventional hierarchic arrangement of FE- and BE-DB is dissolved: If each of them can play both roles and if together they can provide consistency for DB data, more effective DB support may be gained for new applications such as grid or P2P computing.

Furthermore, all caching schemes discussed need careful exploration of their performance behavior. Simulation with analytical models and data using column selectivities and artificial workloads only can provide a rough quantitative overview on costs and benefits. Hence, although very laborious, it must be complemented by empirical performance measurements, at least in a selective way. Interesting questions are related to update frequencies and types, that is, to the performance window in which CBDC is superior to replication or full-table caching. Moreover, how does a mix of different and overlapping predicate extensions perform? When each a single predicate type is mapped to a separate cache table or group, extension overlap would cause replicated data and, in turn, worse modification problems. In addition, for the same BE table, several cache tables would have to be maintained—a design decision which jeopardizes the highly desirable cache transparency for the applications. If at most one cache table is allocated for each BE table, overlapping cache groups, so-called cache group federations [3], necessarily interfere with each other, that is, a cache key may drive cache table filling via RCCs of other cache groups. Of course, some benefit—primarily storage saving—is gained when the same records appear in more than one cache group. However, the penalty may quickly outweigh typically small gains! Many overlapping cache constraints and, in turn, predicate extensions may question the entire approach and call for simple, but communication- and storage-intensive solutions such as full-table caching.

7 Conclusions

We have surveyed the use of CBDC for a variety of query types and primarily have structured this research area top-down. We believe that our definitions are fundamental for describing the related research problems as well as approaching viable solutions. In this respect, the terms value completeness, value range completeness, domain completeness, and predicate completeness are our *magic words*. Because we have explored the underlying concepts of CBDC at the type level, value range completeness and domain completeness to easily control it were most important. However, by controlling value completeness dynamically, CBDC can be improved even further. On the other hand, a combined solution of tasks performed by database caching and such of Web caching may offer additional optimization potential. Moreover, improvement of DB cache adaptivity seems to be a very important issue to make caches (nearly) free of administrative interventions. This aspect of *autonomic computing* is underlined by the fact that today Akamai's content distribution network already has nearly 15 000 edge caching servers [1]. Hence, we are only at the beginning of a promising research area concerning constraint-based and adaptive DB caching where a number of important issues remains to be solved or explored.

References

[1] M. Altinel, Ch. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. Reinwald: Cache Tables: Paving the Way for an Adaptive Database Cache. VLDB Conference 2003: 718–729

[2] K. Amiri, S. Park, R. Tewari, S. Padmanabhan: DBProxy: A Dynamic Data Cache for Web Applications. ICDE Conference 2003: 821–831

[3] T. Härder, A. Bühmann: Value Complete, Domain Complete, Predicate Complete—Magic Words Driving the Design of Cache Groups, submitted (2004)

[4] P.-Å. Larson, J. Goldstein, J. Zhou: MTCache: Mid-Tier Database Caching in SQL Server. ICDE Conference 2004

[5] The TimesTen Team: Mid-tier Caching: The TimesTen Approach. SIGMOD Conference 2002: 588–593

Adaptive Database Caching with DBCache

C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120
Contact email: {mohan, maltinel}@almaden.ibm.com

Abstract

Caching as a means to improve scalability, throughput, and response time for Web applications has received a lot of attention. Caching can take place at different levels in a multi-tier architecture. However, due to the increased personalization and dynamism in Web content, caching at the bottom of the multi-tier architecture – the database level – can be beneficial to all upper levels because of better reusability of the cached data. DBCache supports database caching at both the edge of content delivery networks and at the middle-tier of Web site infrastructures. Its on-demand caching functionality provides the key feature for adaptive and transparent data caching for Web applications.

1 Introduction

Modern e-commerce Web sites, due to their transactional nature and increased personalization, are database driven. As shown in Figure 1, they are typically built on top of a broad range of middleware technologies including network load balancers, Web and application servers, and database servers. System design for a high volume Web site is challenging not only because it involves a variety of different technologies but also because of its performance requirements for peak access times. Scalability can be increased by replicating application server nodes in the middle tier. However, with this design, the backend database tends to become the bottleneck soon. Thus, applying caching solutions for high-volume Web sites to improve scalability, response time, and reliability has received a lot of attention in the Web and the database communities.

DBCache [1] supports the creation of persistent database caches based on DB2's federated query processing capabilities. DBCache allows the specification of *cache tables* that either cache the entire content or a subset of the tables from the backend database server. Federated database capabilities, particularly nicknames, are used to access the backend database. If a query cannot be answered from the cache, it is automatically routed to the backend database. The names of the cache tables can be the same as the backend tables, in which case no changes to existing application queries are required to make use of the cache.

DBCache provides two types of cache tables. *Declarative* cache tables are populated at definition time. They are implemented as user-maintained materialized views. Changes at the backend database are applied to the cache by using a data replication tool. Query routing is done at query compilation time using DB2's materialized view mechanism [9]. *Dynamic* cache tables, on the other hand, are populated at run-time based on the queries issued by the application. Changes at the backend database are detected and applied to the cache based on the respective cache content. Since the content may change over time, query routing is done at query execution time, to take the current content of the cache into consideration. In this paper, we'll focus on dynamic

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

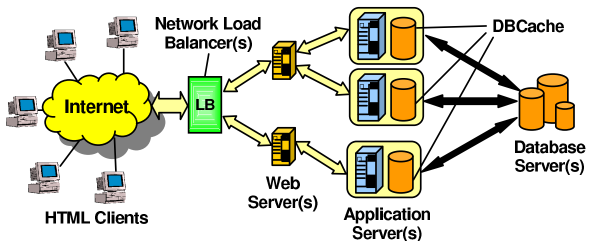

Figure 1: Database Caching with DBCache

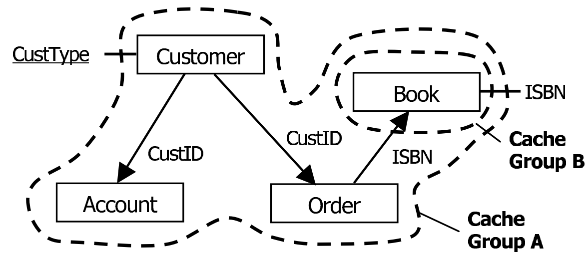

Figure 2: Cache Federation Example

caching and discuss as our main contributions: (1) a model to define a dynamic cache, (2) query plan generation for dynamic caching, and (3) asynchronous mechanisms for cache population and maintenance.

2 The Caching Landscape

Existing caching approaches for Web applications can be broadly classified as Web caching and database caching solutions. In the case of Web caching [7], Web objects such as, entire HTTP pages, page fragments, or JSP results are cached in the Web container of the application servers, the Web servers, a proxy cache, or an existing edge server. With database caching, data from the database server is cached in database caches closer to the application server to offload the backend database server. Even that reuse of complex Web objects may save some processing cost, with increased dynamism and personalization of modern Web sites, database caching offers higher reusability since the same base rows might be used for the generation of different Web objects.

The database cache itself may be implemented in different ways. With Semantic Caching [4] and DBProxy [2] actual query results are cached and new queries are checked to see if they can be satisfied from the cache alone. Query result caching has been proven efficient for complex OLAP type of queries. Another approach is the caching of parts or whole backend tables in a full-fledged database server. Microsoft's MTCache [5] and our previous work in DBCache [6] allow backend-table caching based on explicitly specified materialized views. Unfortunately, it is often impossible to know a priory exactly which table subset to cache because of the dynamic nature of Web applications (e.g., which are the "hot" items from the product catalog?). Thus, with DBCache we allow the specification of dynamic cache tables that cache data based on incoming user queries. In addition, similar to the caching approach in TimesTen [8], we support the declaration of related tables for cache population and maintenance. This way, DBCache offers flexible and adaptive database caching for dynamically changing user access patterns.

3 DBCache Dynamic Caching Model

We have to guarantee that the result of a query using the cache is the same as if it was executed against the backend database[1]. For this, the cache system has to make sure that the complete set of rows required to satisfy a given SQL predicate is completely in the cache. Our current prototype supports the evaluation of equality and equi-join predicates from the cache. We describe the possible content of dynamic cache tables, i.e., the predicates whose extensions should be kept in the cache, by using two types of cache constraints: referential cache constraints and cache keys.

For the evaluation of equality predicates from the cache, domain completeness of the corresponding cache table column is required. *Domain completeness* guarantees that for a given value of the column, the cache table contains all the rows from the backend table with the same value. Note that unique columns are domain-complete

[1] The query result may be slightly different due to the cache being slightly out-of-date (cache latency).

per definition. For non-unique columns, domain completeness needs to be enforced by the caching system. For a single cache table CT, domain completeness of a column c allows for the probing and evaluation of equality predicates of the form $CT.c = v$. If v is found in the cache it is guaranteed that the extension of this predicate is completely in the cache.

To support equi-joins in the cache, information is needed that tells us that if one cache table can be used in the join than the other table can be used too. For this purpose, we introduce the concept of a *referential cache constraint (RCC)*. An RCC defines a cache-parent to cache-child relationship between two join-compatible cache table columns. RCCs are enforced by the caching system. An RCC $CT_i.c_i \rightarrow CT_j.c_j$ guarantees that, when a row r with value v in c_i is in the cache, then all join partners of r via $CT_i.c_i = CT_j.c_j$ are also in the cache. Thus, RCCs allow the evaluation of predicates of the form ($CT_i.c_k = v$ and $CT_i.c_i = CT_j.c_j$), if c_k is domain complete.

The second type of cache constraint, the so-called cache keys, specify how cache tables are populated. Declaring a cache table column as a cache key assigns the domain completeness property to it. In addition, a cache key specifies a filling point for the cache table and, via (a chain of) RCCs, for other cache tables related to it. When a user query in an equality predicate refers a cache key value v which is currently not in the cache, evaluation of this predicate has to be done at the backend. As a consequence of such a cache miss, the caching system brings in all required rows for v from the backend database. The assumption here is that because of locality of reference a given cache key value indicates the likely reuse of the inserted rows.

A set of related cache tables which are (directly or via RCCs) populated by the values of one or more cache keys of a single cache table is called a *cache group*. A cache group can be understood as the representation of the processing context of one or more query types. Multiple cache groups may (partially) overlap in one or more cache tables thereby establishing a so-called *cache group federation*. Figure 2 shows an example of a federation consisting of two overlapping cache groups. Cache group A is filled through cache key *CustType* and contains all cache tables depicted. Cache group B is filled through *ISBN* and contains only the *Book* cache table.

Unrestricted use of cache constraints may result in performance and scalability problems because of cache population and cache maintenance overhead. Thus, we provide the following design rules and recommendation. Excessive cache population occurs when we have recurring load operations for the same cache table, which in the extreme case may result in caching the entire backend table. Obviously, cycles created by RCC definitions[2] in cache groups may pose this *recursive cache load* problem. We distinguish two types of cycles. In a *homogeneous* cycle for each participating table only one column is involved in the RCC cycle. A *heterogeneous* cycle is formed if one of the tables contains two columns used in the cycle. Homogeneous cycles are safe w.r.t. the recursive cache load problem. This is because starting with a given set of values a traversal does not require additional values for the participating columns. By the same argument, heterogeneous cycles are not safe since tables with two columns involved may require additional values for the column we started off from and, in turn, enforce an additional circulation. Therefore, as our first design rule, we do not allow heterogeneous cycles in cache group definitions.

In our cache model, a cache table column becomes domain complete if: (1) it is a unique (or primary key) column, (2) a cache key is defined for it, (3) it is involved in a homogeneous cycle[3], or (4) it is the target of the only incoming RCC and its table does not contain a cache key defined on another column (aka *induced domain completeness*). Enforcing domain-completeness for non-unique columns may lead to recursive cache population. Let's assume a cache table with two non-unique domain complete columns c_1 and c_2. If we insert rows for a new value for c_1, to ensure domain completeness of c_2 we need to insert more rows for each new value in c_2, which in turn may require to bring in new rows to ensure domain completeness of c_1 and so forth. Thus, as our second design rule, we allow at most one non-unique domain-complete column per cache table. However, we pose no limit on the number of unique domain-complete columns.

[2]We define a cycle as a path from a cache table back to the same table by following a set of RCCs. At this point, each participating table must only be visited once.

[3]For the initiating column, i.e., the column through which the cycle is populated domain completeness is enforced by the caching system. For all other columns in the cycle induced domain completeness applies.

Low-selectivity cache key columns (e.g., gender) can trigger the loading of large amounts of data that is never used later. Thus, the use of high-selectivity cache keys (e.g., customer ID) is recommended which allows a more precise population of the cache.

4 Janus Query Plans

Since the content of dynamic cache tables is not described by any predicate values and may change between subsequent executions of the same query, the decision of whether to route the query to the cache or the backend database must be made at runtime. For this, a special kind of query execution plan called *Janus* plan is prepared at query compilation time. A Janus plan consists of two query plan alternatives. The *local* plan refers to all dynamic cache tables that, dependent on the current content, might be usable for query execution. For tables for which no cache table exists, the local plan refers via a nickname to the corresponding backend table.[4] The *remote* plan, on the other hand, uses only nicknames to enable execution of the query at the backend database. Both query plans are tied together with a conditional switch operator which contains a simple subquery, called the *probe* query. Usability of cache tables in the probe and the local query plan is decided based on cache constraints. Details are given in [1].

Execution of a Janus plan is done by first executing the probe query which performs a simple existence check for the cache tables referred to in the local query to find out whether or not the local plan can be used to answer the given query. If the cache contains the necessary data the local query plan is executed, if not we fall back to the remote query plan. The probe query is constructed based on the equality predicates on domain complete columns as given in the user query. The property of domain completeness, as introduced in Section 3, ensures that the probe is sufficient to guarantee correctness – if a single record of the probe query result is found in the cache, it is guaranteed that it is safe to execute the local query plan.

The probe query is created as a scalar subquery and its results can potentially be reused by the local query. Thus, as shown in [1], the extra overhead of the probe query is relatively low and justified by the benefits of executing a local as opposed to a remote query.

5 Cache Population

Each execution of the remote query in a Janus plan is considered a cache miss, and the cache key values that failed the probe query are used for on-demand cache population by a dedicated background process called the DBCache cache daemon.[5] Cache miss information is passed on to the cache daemon by sending an internal message as part of remote query plan execution. However, cache tables are not populated as part of the remote query. This is because cache constraints may require the loading of a (relatively) large amount of additional data into a number of cache tables; also it would implicitly turn the given user query into an insert statement. Both would significantly increase execution time and change the semantics of the original user query.

For each cache miss message, the daemon generates (at most) one insert statement per cache table in the affected cache group and then executes these statements in a single database transaction. As a first step for statement generation, the set of *qualifying rows* for each cache table, i.e., the rows that need to be brought in because of defined cache constraints. Qualifying rows for a cache table are represented as (nested) subqueries that refer to the corresponding backend tables, is determined. Qualifying rows for the root table CT_0 of the cache group are determined by the given cache key value and cache keys defined on CT_0. As we follow RCCs $CT_i.c_{from} \rightarrow CT_{i+1}.c_{to}$ the qualifying rows subquery for CT_i is used as the starting point to determine the qualifying rows for CT_{i+1}. The final insert statement for each cache table is based on the qualifying rows for the

[4] Note that this may result in a distributed executed of the query using DB2's federated database functionality.

[5] Here the execution of application-specific policies is possible, e.g., "only start caching if we have seen the same cache miss n times in the last m seconds".

```
INSERT INTO UserSchema.Customer
SELECT * FROM NN.Customer
WHERE CustType = 'gold'

INSERT INTO UserSchema.Account
SELECT * FROM NN.Account
WHERE CustID IN (SELECT CustID
                 FROM NN.Customer
                 WHERE CustType = 'gold')

INSERT INTO UserSchema.Order
SELECT * FROM NN.Order
WHERE CustID IN (SELECT CustID
                 FROM NN.Customer
                 WHERE CustType = 'gold')
  AND (OrdID) NOT IN (SELECT OrdID FROM
                      UserSchema.Order)

INSERT INTO UserSchema.Book
SELECT * FROM NN.Book
WHERE ISBN IN (SELECT ISBN FROM NN.Order
               WHERE CustID IN
                 (SELECT CustID
                  FROM NN.Customer
                  WHERE CustType = 'gold'))
```

Figure 3: Insert Statement Generation

```
DELETE FROM UserSchema.Order
WHERE CustID IN (SELECT CustID
                 FROM UserSchema.Customer
                 WHERE CustID = 'cid:500')
  AND (ISBN) NOT IN (SELECT ISBN
                     FROM UserSchema.Book)

DELETE FROM UserSchema.Account
WHERE CustID IN (SELECT CustID
                 FROM UserSchema.Customer
                 WHERE CustID = 'cid:500')

DELETE FROM UserSchema.Customer
WHERE CustID = 'cid:500'
```

Figure 4: Delete Statement Generation

respective table and results in a federated statement. This way, the daemon does not have to deal with intermediary query results, i.e., actual rows. For our example from Figure 2, a cache miss for *CustType = "gold"* would result in the insert statements shown in Figure 3.

Cache miss messages need not be persistent. If a pending miss message is lost, another miss message for the same cache key value is sent. We require a primary key for each cache table. Thus, if a cache miss message is sent multiple times (i.e., because a second request for the same cache key occurred before the first cache miss was processed) the resulting primary key constraint violation causes the daemon to skip the second message.

6 Cache Maintenance

The content of dynamic cache tables needs to be updated as the corresponding tables at the backend database are updated. Changes at the backend are detected by the capture program of IBM's data replication tool which sends according maintenance messages to the DBCache cache daemon. All changes from a transaction at the backend are sent as one message to preserve transaction boundaries. Also, since losing maintenance messages might result in inconsistent cache content, messages from the replication tool have to be stored persistently.

Upon receiving a maintenance message, the daemon generates appropriate insert, update and delete statements and issues them against the cache database. The algorithms used to generate these statements are similar to the one described for cache population in Section 5. All changes of a transaction at the backend database are applied to the cache within the same database transaction. Note that changes at the backend tables might be relevant for the cache database or not based on the current cache content. In our existing prototype, filtering out irrelevant changes is done by the daemon during statement execution. In particular, the following four cache maintenance tasks are performed by the daemon.

Insert at the backend database: A new row has to be inserted into the cache only if (a) for any value of a column that is the target of an incoming RCC we have the corresponding cache parent row(s) in the cache, or if (b) for a non-unique cache key of the target table we already have rows with the same key value in that table. Otherwise, the insert at the backend database is irrelevant for the cache database. This way, we only bring in new cache key values based on an insert at the backend if existing cache constraints require us to do so. If the daemon tries to insert an already existing row, (e.g., because a cache miss message overtook a cache maintenance message[6]), the resulting primary key violation causes the daemon to skip the maintenance message. The insert statement for the root table is based on the insert at the backend database and the defined cache constraints. For all other cache tables statement generation is done exactly as for cache population.

Update at the Backend Database. For update operations we distinguish two cases. (a) If the update is only on columns that are neither the source or target of an RCC nor cache key columns we can do the update by simply updating the corresponding row in the cache table. (b) If the update is on at least one cache key or RCC column we might have to delete and/or insert other rows from/into the cache in addition to the actual update. For the latter case, we can translate the update into a delete of the old row, using the algorithm for backend deletes, followed by an insert of the updated row, using the algorithm for backend inserts. If an update message is received by the daemon before the affected row was brought into the cache (because of a cache miss) the update message is discarded. This is not a problem since at the time the corresponding cache miss message is processed the latest version (i.e., the one reflecting the update seen before) of the row is fetched.

Deletion from the Backend Database. If a cached row is deleted from the backend database, we also have to remove this row from the corresponding cache table. In the case of a cache parent row, we actually do not need to delete all corresponding cache children to preserve cache consistency. However, in spite of this, we also delete cache children if (a) the cache child table does not have a cache key, and (b) the cache children are not shared by a cache parent of the same or another cache group. Since garbage collection is done through cache keys, these rows would become unreachable for later garbage collection. If a delete message is seen by the daemon before the corresponding cache miss message that would bring in the affected row, the delete message will have no effect. This does not cause a problem since at the time the cache miss message is processed the deleted row would simply not be found at the backend database. Similar to statement generation for inserts, the main idea behind delete statement generation is to generate one delete statement per cache table, and then to execute these statements in a single database transaction. The delete statement for a given cache table is generated based on the non-unique cache keys defined on this table and the qualifying rows identified by the delete message or the qualifying rows of its cache parent tables. For our example from Figure 2, the deletion of customer row *CustID = ”cid:500”* would result in the delete statements shown in Figure 4. Note that we do not generate a delete statement for the *Book* cache table since it has a cache key defined.

Garbage Collection. Periodic garbage collection done by the cache daemon directly deletes rows from the cache based on cache key values determined to be ”good” candidates for eviction. The purpose is to delete entire cache group instances (”trees” of related rows) to free up caching space. Identification of candidates for eviction can be based on popularity (i.e., how many cache hits did occur for this cache group instance), recency (i.e., how long has the cache group instance been in the cache), and space consumption (i.e., how big is the cache group instance). Statement generation for garbage collection is done in a very similar way as for backend deletes with only the following two differences. (a) In case of the eviction from a root table that has a non-unique cache key, we have to delete *all* rows that have the corresponding cache key value to conserve domain completeness. This is true even if the eviction is not for a value of the non-unique cache key. (b) For garbage collection and backend deletes alike, if deletion of a cache parent is required we also delete all corresponding cache children that are not referred to by a cache parent of another cache group. For garbage collection, however, the only exception is if the cache group that shares child rows is fully contained in the group for which we perform deletes. In this case we also delete the shared rows which results in a more aggressive deletion for garbage collection. From the cache daemon’s point of view, garbage collection corresponds to backend deletes. Thus, conflicts based on different execution orders of different maintenance operations are not an issue for the reasons given in the previous paragraphs.

[6] In the current prototype all incoming messages for the same cache federation are processed by the daemon in sequential order to guarantee cache consistency and to reduce the likelihood of deadlocks.

7 Open Research Issues

In this section we will identify open research issues in the area of dynamic database caching. Although the issues are discussed in a DBCache-specific context, they pose problems relevant to constraint-based sub-table caching in general.

Cache Maintenance – Update Filtering. Whether a given update at the backend database is relevant for a particular cache depends on the current cache content. To determine which changes at the backend need to be applied to which caches, we have considered two architectural alternatives. (a) In the first case, capture sends a maintenance message for every update at the backend database to every frontend cache, and the respective cache daemon determines which operations to be apply to the cache database[7]. Since this approach, in general, can introduce significant message traffic between the backend and the cache database, it might only be applicable to applications with very low update ratio, or to settings with only a few frontend caches and with high network bandwidths. (b) In the second case, determining which update operations need to be applied at which cache is done at the backend database. Thus, only relevant maintenance messages are sent from the backend to the affected caches and no additional filtering at the frontends is required. This approach requires a complete representation of cache constraints and keeping close track of the cache content at the backend database for every cache served. It might be beneficial for applications with higher update ratio by significantly reducing the message traffic. However, it requires additional processing at the backend database hampering our goal of unburdening the backend server. Although for a complete implementation an adaptive solution that shifts filtering between cache and backend database nodes is desirable, the following two main questions remain open. First, how can we organize and maintain information at the backend about the current cache contents without overwhelming the backend server? Second, based on a given update ratio, cache sizes, and number of caches, which of the two alternatives is superior?

Local Updates. In the initial prototype, all update queries are executed at the backend database and are applied indirectly to the cache through the cache daemon. As a result, applications do not see the effects of their updates within the same transaction. This may not be acceptable for certain classes of applications. We distinguish three update modes: (a) direct update of backend database, indirect update of local cache (via replication tool), (b) direct update of backend and local cache (requires 2PC support), and (c) direct update of local cache, indirect update of backend database (via replication tool). A database caching system has to offer these different update options so that applications can choose the best fit for their update semantics. Moreover, there is a need for new mechanisms for detecting whether an update operation is relevant to the local cache. There is also a need to change probing logic in the Janus plans. The reason is that due to effects of update operations (i.e. deleting a row), checking content of cache tables in the probe may select the wrong plan option. Hence, applications may end up not seeing the effects of their updates. One solution to this problem is to probe some form of metadata instead of actual table content. Open questions in this case are, what kind of metadata would be needed to represent the cache content? How this information can be kept up-to-date with respect to dynamically changing cache content? What kind of race conditions may arise and how can they be avoided?

Garbage Collection. To preserve cache consistency, garbage collection for DBCache has to be done at cache group instance level and not at the level of individual rows. This poses two major challenges. (a) Cached objects, i.e., cache group instances, are of varying size; and (b) different from most Web object caching scenarios, dependencies between cached objects (in the form of overlapping cache groups) have to be taken into consideration. For different object sizes we can retreat to existing function-based strategies as used for Web object caching [7]. For overlapping cache groups expensive filtering is required to prevent deletion of rows shared by multiple cache group instances. Main open issues for garbage collection in the context of cache groups are:

[7] Our current prototype follows this approach.

First, since eviction is based on statistics like number of cache hits or size of cache group instances, what would be a reasonable trade-off between accuracy and cost for collecting necessary statistics? Second, in the presence of overlapping cache groups and cache maintenance operations, how can we efficiently attribute cache hits (i.e., the use of rows from the cache to answer a given query) to cache group instances?

Tooling. Efficient use of constraint-based database caching requires the provision of powerful tools to support cache setup and adaptation. We envision the development of a cache advisor tool to propose and automatically setup cache tables and cache constraints based on a given query workload. If no such workload can be provided, the tool can still automate the process by taking hints from the backend database schema, i.e., primary keys and referential integrity constraints can be a good starting point for selecting cache keys and RCCs. In addition, tooling is required to adapt a given cache setup at run-time in response to changed user behavior by adding or dropping cache elements. Close monitoring of cache performance is required to support well informed manual corrections. Open issues in this area include the question for appropriate cost models to estimate cache population/maintenance cost vs. benefit for different cache setup alternatives. These models form the basis for automatic decisions on cache setup, and thus on-demand database caching.

8 Summary

DBCache dynamic cache tables provide adaptive on-demand database caching for high-volume e-commerce Web sites. In this paper, we briefly discussed our main contributions: (1) a model to define dynamic caches, (2) query plan generation for dynamic caching, and (3) asynchronous mechanisms for cache population and maintenance. For a more detailed discussion of the dynamic cache model, and DBCache query plan generation see [1]. We would like to thank Nesime Tabul, Prakash Linga, Sailesh Krishnamurthy, Bruce Lindsay, Dan Wolfson, and Theo Haerder for their contributions to the DBCache project, and the IBM SWG for many fruitful discussions.

References

[1] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. Reinwald: *Cache Tables: Paving the Way for an Adaptive Database Cache*, VLDB'03, Berlin, Germany, Sep. 2003

[2] K. Amiri, S. Park, R. Tewari, S. Padmanabhan: *DBProxy: A Dynamic Data Cache for Web Applications*, ICDE'03, Bangalore, India, Mar. 2003

[3] C. Bornhövd, M. Altinel, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. Reinwald: *DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures*, (Demo Description), SIGMOD'03, San Diego, CA, Jun. 2003

[4] S. Dar, M. Franklin, B. Jonsson, D. Srivastava, M. Tan: *Semantic Data Caching and Replacement*, VLDB'96, Mumbai (Bombay), India, Sep. 1996

[5] P. Larson, J. Goldstein, J. Zhou: *Transparent Mid-tier Database Caching in SQL Server*, (Demo Description), SIGMOD'03, San Diego, CA, Jun. 2003

[6] Q. Luo, S. Krishnamurthy, C. Mohan, H. Woo, H. Pirahesh, B. Lindsay, J. Naughton: *Middle-tier Database Caching for e-Business*, SIGMOD'02, Madison, WI, Jun. 2002

[7] S. Podlipnig, L. Böszörmenyi: *A Survey of Web Cache Replacement Strategies*, ACM Computing Surveys, Vol. 35, No. 4, Dec. 2003, pp. 374-398

[8] The TimesTen Team: *Mid-tier Caching: The FrontTier Approach*, SIGMOD'02, Madison, WI, Jun. 2002

[9] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, M. Urata: *Answering Complex SQL Queries Using Automatic Tables*, SIGMOD'00, Philadelphia, PA, May 2000

C-JDBC: a Middleware Framework for Database Clustering

Emmanuel Cecchet
INRIA Rhône-Alpes - Sardes team
655, Avenue de l'Europe - 38330 Montbonnot - France
emmanuel.cecchet@inria.fr

Abstract

Clusters of workstations become more and more popular to power data server applications such as large scale Web sites or e-Commerce applications. Successful open-source tools exist for clustering the front tiers of such sites (web servers and application servers). No comparable success has been achieved for scaling the backend databases. An expensive SMP machine is required if the database tier becomes the bottleneck. The few tools that exist for clustering databases are often database-specific and/or proprietary.

Clustered JDBC (C-JDBC) addresses this problem. It is an open-source, flexible and efficient middleware for database clustering. C-JDBC implements the Redundant Array of Inexpensive Databases (RAIDb) concept. It presents a single virtual database to the application through the JDBC interface and does not require any modification to existing applications. Furthermore, C-JDBC works with any database engine that provides a JDBC driver, without modification to the database engine. The C-JDBC framework is open, configurable and extensible to support large and complex database cluster architectures offering various performance, fault tolerance and availability tradeoffs.

1 Introduction

Nowadays, database scalability and high availability can be achieved, but at very high expense. Existing solutions require large SMP machines or clusters with a Storage Area Network (SAN) and high-end RDBMS (Relational DataBase Management Systems). Both hardware and software licensing cost makes those solutions only available to large businesses. Commercial implementations such as Oracle Real Application Clusters [12] that address cluster architectures requires a shared storage system such as a SAN (Storage Area Network). Open-source solutions for database clustering are database-specific. MySQL cluster [13] uses a synchronous replication mechanism with limited support for scalability (up to 4 nodes). Some experiments have been reported using partial replication in Postgres-R [8]. These extensions to existing database engines often require applications to use additional APIs to benefit from the clustering features. Moreover, these different implementations do not interoperate well with each other.

Database replication has been used as a solution to improve availability and performance of distributed or clustered databases [1, 9]. Even if many protocols have been designed to provide data consistency and fault tolerance [4], few of them have found their way into practice [14]. Gray et al. [10] have pointed out the danger

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

of replication and the scalability limit of this approach. However, database replication is a viable approach if an appropriate replication algorithm is used [2, 8, 15]. Most of these recent works only focus on full database replication.

We have introduced Redundant Array of Inexpensive Databases or RAIDb [5] as a classification of different database replication techniques. RAIDb is an analogy to the existing RAID (Redundant Array of Inexpensive Disks) concept, that achieves scalability and high availability of disk subsystems at a low cost. RAID combines multiple inexpensive disk drives into an array of disk drives to obtain performance, capacity and reliability that exceeds that of a single large drive [7]. RAIDb is the counterpart of RAID for databases. RAIDb aims at providing better performance and fault tolerance than a single database, at a low cost, by combining multiple database instances into an array of databases. RAIDb primarily targets low-cost commodity hardware and software such as clusters of workstations and open source databases.

We have implemented Clustered JDBC (C-JDBC), a software implementation of RAIDb. C-JDBC is an open-source Java middleware framework for database clustering on a shared-nothing architecture built with commodity hardware. C-JDBC implements the RAIDb concepts hiding the complexity of the cluster and offering a single database view to the application. The client application does not need to be modified and transparently accesses a database cluster as if it were a centralized database. C-JDBC works with any RDBMS that provides a JDBC driver. The RDBMS does not need any modification either, nor does it need to provide distributed database functionalities. Load distribution, fault tolerance and failure recovery are all handled at the middleware level by C-JDBC. The architecture is flexible, distributable and extensible to support large clusters of heterogeneous databases with various degrees of performance, fault tolerance and availability.

With C-JDBC, we hope to make database clustering available in a low-cost and powerful manner, thereby spurring its use in research and industry. Although C-JDBC has only been available for a year, several installations are already using it to support various database clustering applications [6]. The outline of the rest of this paper is as follows. Section 2 introduces the RAIDb concepts. Section 3 presents the architecture of C-JDBC and the role of its internal components. Section 4 describes how fault tolerance is handled in C-JDBC and section 5 discusses horizontal and vertical scalability. We conclude in section 6.

2 Redundant Array of Inexpensive Databases

One of the goals of RAIDb is to hide the distribution complexity and provide the database clients with the view of a single database like in a centralized architecture. As for RAID, a controller sits in front of the underlying resources. Clients send their requests directly to the RAIDb controller that distributes them among the set of RDBMS backends. The RAIDb controller gives the illusion of a single RDBMS to the clients.

RAIDb does not impose any modification of the client application or the RDBMS. However, some precautions have to be taken care of, such as the fact that all requests to the databases must be sent through the RAIDb controller. It is not allowed to directly issue requests to a database backend as this might compromise the data synchronization between the backends.

RAIDb defines 3 basic levels providing various degree of replication that offer different performance/fault tolerance tradeoffs.

2.1 RAIDb-0: full partitioning

RAIDb level 0 consists in partitioning the database tables among the nodes. It does not duplicate information and therefore does not provide any fault tolerance guarantees. RAIDb-0 allows large databases to be distributed, which could be a solution if no node has enough storage capacity to store the whole database. Also, each database engine processes a smaller working set and can possibly have better cache usage, since the requests are always hitting a reduced number of tables.

2.2 RAIDb-1: full replication

With RAIDb level 1, databases are fully replicated. RAIDb-1 requires each backend node to have enough storage capacity to hold all database data. RAIDb-1 needs at least 2 database backends, but there is (theoretically) no limit to the number of RDBMS backends. The performance scalability is bounded by the capacity of the RAIDb controller to efficiently broadcast the updates to all backends.
RAIDb-1 provides speedup for read queries because they can be balanced over the backends. Write queries are performed in parallel by all nodes, therefore they usually execute at the same speed as the one of a single node. However, RAIDb-1 provides good fault tolerance, since it can continue to operate with a single backend node.

2.3 RAIDb-2: partial replication

RAIDb level 2 features partial replication which is an intermediate solution between RAIDb-0 and RAIDb-1. Unlike RAIDb-1, RAIDb-2 does not require any single node to host a full copy of the database. This is essential when the full database is too large to be hosted on a node's disks. Each database table must be replicated at least once to survive a single node failure. RAIDb-2 uses at least 3 database backends (2 nodes would be a RAIDb-1 solution).

3 C-JDBC: a RAIDb software implementation

JDBC, often referenced as Java Database Connectivity, is a Java API for accessing virtually any kind of tabular data [19]. C-JDBC (Clustered JDBC) is a Java middleware based on JDBC, that allows building all RAIDb configurations described in the previous section.

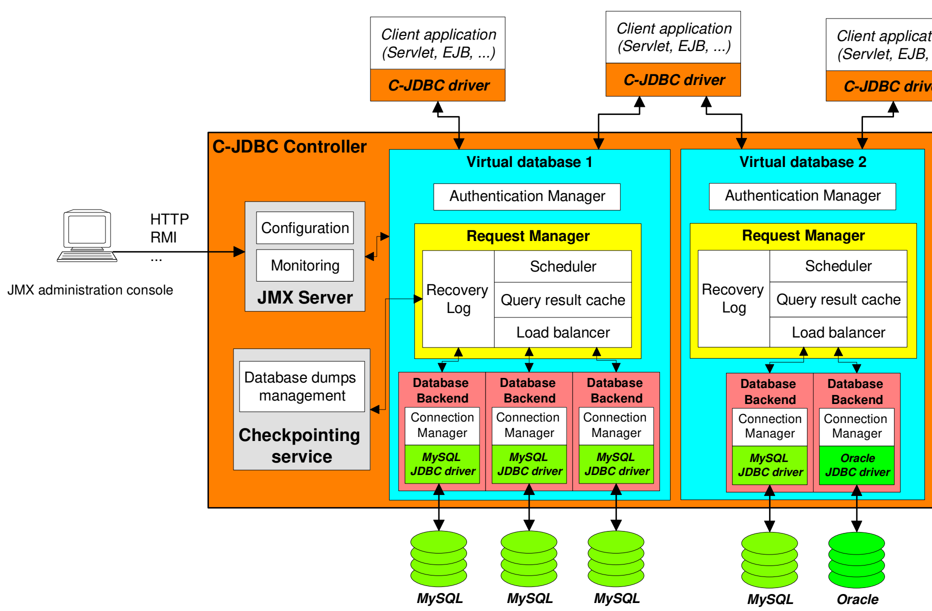

Figure 1: C-JDBC architecture overview.

3.1 Architecture overview

C-JDBC is made of 2 components: a generic JDBC driver to be used by the client application and a controller that handles load balancing and fault tolerance. The database-specific JDBC driver used by the client application is replaced by a generic C-JDBC driver that offers the same interface. The C-JDBC controller is a Java program that acts as a proxy between the C-JDBC driver and the database backends. The distribution is handled by the C-JDBC controller that implements the logic of a RAIDb controller. The degree of replication and the location of the replicas can be specified on a per-table basis. Currently, the tables named in a particular query must all be present on at least one backend. In dynamic content servers, one of the target environments for C-JDBC clusters, this requirement can often be met, since the queries are known ahead of time. Eventually, we plan to add support for distributed execution of a single query.

The C-JDBC controller exposes a single database view, called a virtual database, to the C-JDBC driver and thus to the application. A virtual database has a virtual name that matches the database name used in the client application. Virtual login names and passwords match also the ones used by the client. The client therefore perceives there to be a single (virtual) database. Database back-ends can be dynamically added to or removed from a virtual database, transparently to the user applications.

An authentication manager establishes the mapping between the login/password provided by the client application (called virtual login) and the login/password to be used on each database backend (called real login). This allows database backends to have different names and user access rights mapped to the same virtual database setting. All security checks can be performed by the authentication manager. It provides a uniform and centralized resource access control.

A controller can possibly host multiple virtual databases but each virtual database has its own request manager that defines its request scheduling, caching and load balancing policies. Routing of queries to the various backends is done automatically by C-JDBC, using a read-one write-all approach. A number of strategies are available for load balancing and connection pooling, with the possibility of overriding these strategies with a user-defined one. C-JDBC can be configured to support result caching and fault tolerance.

Finally, larger and more highly-available systems can be built by using the horizontal and vertical scalability features of C-JDBC (see section 5). C-JDBC also provides additional services such as monitoring and logging. The controller can be dynamically configured and monitored through JMX (Java Management eXtensions) either programmatically or using an administration console.

3.2 C-JDBC driver

The C-JDBC driver implements the JDBC 2.0 specification and some extensions of the JDBC 3.0 specification. Queries are sent to the controller that issues them on a database backend. The result set is serialized by the controller into a C-JDBC driver `ResultSet` that can be sent through the communication channel. The driver ResultSet contains the logic to browse the results locally to the client machine without requiring further interaction with the controller.

When the ResultSet is too large to be sent at once due to memory constraints, it is fetched by blocks of fixed number of rows. These blocks are transparently sent to the driver when they are required.

The C-JDBC driver can also transparently fail over between multiple C-JDBC controllers, implementing horizontal scalability (see Section 5). The JDBC URL used by the driver is made up of a comma-separated list of 'node/port' items, for instance, `jdbc:cjdbc://node1,node2/db` followed by the database name. When the driver receives this URL, it randomly picks up a node from the list. This allows all client applications to use the same URL and dynamically distribute their requests on the available controllers.

3.3 Request manager

Incoming requests are routed to the request manager associated with the virtual database. The request manager contains the core logic of the virtual database. It is composed of a scheduler, optional caches, a load balancer and a recovery log. Each of these components can be superseded by a user-specified implementation.

Scheduler The *scheduler* is responsible for handling the concurrency control and for ordering the requests according to the desired isolation level. C-JDBC provides both optimistic and pessimistic transaction management. Transaction demarcation operations (begin, commit and rollback) are sent to all backends. Reads are sent to a single backend. Updates are sent to all backends where the affected tables reside. Depending on the RAIDb level, this may be one (RAIDb-0), several (RAIDb-2) or all (RAIDb-1) backends. SQL queries containing macros such as RAND() or NOW() are rewritten on-the-fly with a value computed by the scheduler so that each backend stores exactly the same data.
All operations are synchronous with respect to the client. The request manager waits until it has received responses from all backends involved in the operation before it returns a response to the client. To improve performance, C-JDBC also implements parallel transactions and early response to update, commit, or abort requests. With parallel transactions, operations from different transactions can execute at the same time on different backends. Early response to update, commit or abort allows the controller to return the result to the client application as soon as one, a majority or all backends have executed the operation. Returning the result when the first backend completes the command offers the latency of the fastest backend to the application. When early response to update is enabled, C-JDBC makes sure that the order of operations in a single transaction is respected at all backends. Specifically, if a read follows an update in the same transaction, that read is guaranteed to execute after the update has executed.
If a backend executing an update, a commit or a rollback fails, it is disabled. In particular, C-JDBC does not use a 2-phase commit protocol. Instead, it provides tools to automatically recover failed backends into a virtual database (see Section 4).
At any given time only a single update, commit or abort is in progress on a particular virtual database. Multiple reads from different transactions can be going on at the same time. Updates, commits and aborts are sent to all backends in the same order.

Caches C-JDBC provides 3 optional caches. The *parsing cache* stores the results of query parsing so that a query that is executed several times is parsed only once. The *metadata cache* records all ResultSet metadata such as column names and types associated with a query result.
These caches work with query skeletons found in `PreparedStatements` used by application server. A query skeleton is a query where all variable fields are replaced with question marks and filled at runtime with a specific API. An example of a query skeleton is `SELECT * FROM t WHERE x=?`. In this example, a parsing or metadata cache hit will occur for any value of `x`.
The *query result cache* is used to store the `ResultSet` associated with each query. The query result cache reduces the request response time as well as the load on the database backends. By default, the cache provides strong consistency. In other words, C-JDBC invalidates cache entries that may contain stale data as a result of an update query. Cache consistency may be relaxed using user-defined rules. The results of queries that can accept stale data can be kept in the cache for a time specified by a staleness limit, even though subsequent update queries may have rendered the cached entry inconsistent.
We have implemented different cache invalidation granularities ranging from database-wide invalidation to table-based or column-based invalidation. An extra optimization concerns queries that select a unique row based on a primary key. These queries are often issued by application servers using JDO (Java Data Objects) or EJB (Enterprise Java Beans) technologies. These entries are never invalidated on inserts since a newly inserted row will always have a different primary key value and therefore will not affect this kind of cache entries. Moreover, update or delete operations on these entries can be easily performed in the cache.

Load balancer If no cache has been loaded or a cache miss has occurred, the request arrives at the *load balancer*. C-JDBC offers various load balancers for the different RAIDb levels.
RAIDb-1 is easy to handle. It does not require request parsing since every database backend can handle any

query. Database updates, however, need to be sent to all nodes, and performance suffers from the need to broadcast updates when the number of backends increases.
RAIDb-0 or RAIDb-2 load balancers need to know the database schema of each backend to route requests appropriately. The schema information is dynamically gathered. When the backend is enabled, the appropriate methods are called on the JDBC `DatabaseMetaData` information of the backend native driver. Database schemas can also be statically specified by the way of the configuration file. This schema is updated dynamically on each *create* or *drop* SQL statement to reflect each backend schema.
Among the backends that can treat the request (all of them in RAIDb-1), one is selected according to the implemented algorithm. Currently implemented algorithms are round robin, weighted round robin and least pending requests first (the request is sent to the node that has the least pending queries).
In the case of an heterogeneous cluster (database engines from different vendors), it is possible to define rewriting rules based on patterns to accommodate the various SQL dialects. These rules are defined on a per-backend basis and allow queries to be rewritten on-the-fly to execute properly at each backend.

4 Fault tolerance

To achieve fault tolerance, database content must be replicated on several nodes. C-JDBC uses an ETL (Extraction Transforming Loading) tool called Octopus [11] to copy data to/from databases.

Building the initial state The initial database is dumped (data and metadata) on the filesystem in a portable format. We call this the initial checkpoint. Octopus takes care of re-creating tables and indexes using the database specific types and syntax.
Once the system is online, a *recovery log* records all SQL statements that update the database. When a new backend is added to the cluster, the initial checkpoint is restored on the node using Octopus and the recovery log replays all updates that occurred since the initial checkpoint. Once this new backend is synchronized with the others, it is enabled to server client requests.

Checkpointing At any point in time, it is possible to perform a checkpoint of the virtual database, that is to say a dump of the database content. Checkpointing can be manually triggered by the administrator or automated based on temporal rules.
Taking a snapshot from a backend while the system is online requires to disable this backend so that no update occur on it during the backup. The other backends remain enabled to answer the client requests. As the whole database needs to be consistent, trying to backup a backend while leaving it enabled would require to lock all tables in read and thus blocking all writes. This is not possible when dealing with large databases where copying the database content may take hours.
Disabling a specific backend for checkpointing uses a specific data path that does not load the system. At the beginning of the backup, a specific index is inserted in the recovery log (see below). Once the database content has been successfully dumped, the updates that occurred during the backup are replayed from the recovery log on the backend that was used for checkpointing. Once the backend is synchronized with the others, it is enabled again.

Recovery log C-JDBC implements a recovery log that records a log entry for each begin, commit, abort and update statement. A log entry consists of the user identification, the transaction identifier, and the SQL statement. The log can be stored in a flat file, but also in a database using JDBC. A fault-tolerant log can then be created by sending the log updates to a virtual C-JDBC database with fault tolerance enabled.

5 Horizontal and vertical scalability

As mentioned in section 3.2, *horizontal scalability* provides C-JDBC controller replication to prevent the controller from being a single point of failure. To support a large number of database backends, we also provide *vertical scalability* to build a hierarchy of backends. To deal with very large configurations where both high availability and high performance are needed, one can combine horizontal and vertical scalability [6].

5.1 C-JDBC horizontal scalability

Horizontal scalability is what is needed to prevent the C-JDBC controller from being a single point of failure. We use the JGroups [3] group communication library to synchronize the request managers of the virtual databases that are distributed over several controllers.
When a virtual database is loaded into a controller, a group name is assigned to the virtual database. This group name is used to communicate with other controllers hosting the same virtual database. At initialization time, the controllers exchange their respective backend configurations. If a controller fails, a remote controller can recover the backends of the failed controller using the information gathered at initialization time.
C-JDBC relies on JGroups' reliable and totally ordered message delivery to synchronize write requests and demarcate transactions. Only the request managers contain the distribution logic and use group communication. All other C-JDBC components (scheduler, cache, and load balancer) remain the same.

5.2 C-JDBC vertical scalability

As a C-JDBC controller gives the view of a single database to the client application, it is possible to implement the backends of a C-JDBC controller with other C-JDBC controllers. In general, an arbitrary tree structure can be created. The C-JDBC controllers at the different levels are interconnected by C-JDBC drivers. The native database drivers connect the leaves of the controller hierarchy to the real database backends.
Vertical scalability may be necessary to scale an installation to a large number of backends. Limitations in current Java Virtual Machines restrict the number of outgoing connections from a C-JDBC driver to a few hundreds. Beyond that, performance drops off considerably. Vertical scalability spreads the number of connections over a number of JVMs, retaining good performance.

6 Conclusion

Clustered JDBC (C-JDBC) is a flexible and extensible middleware framework for database clustering, addressing high availability, heterogeneity and performance scalability. C-JDBC primarily targets J2EE application servers requiring database clusters build with commodity hardware. By using the standard JDBC interface, C-JDBC remains database vendor independent and it is transparent to JDBC applications. Combining both horizontal and vertical scalability provide support for large-scale replicated databases. Several experiments have shown that C-JDBC caches improve performance further even in the case of a single database backend.
There is a growing academic and industrial community that shares its experience and provides support on the `c-jdbc@objectweb.org` mailing list. C-JDBC is an open-source project licensed under LGPL hosted by the ObjectWeb consortium. C-JDBC is available for download from `http://c-jdbc.objectweb.org`.

Acknowledgements

We are grateful for the C-JDBC users community for their feedback and support. We would like to thank all C-JDBC contributors who help us improve the software by testing it in various environments and contribute to the code base with fixes and new features.

References

[1] Christiana Amza, Alan L. Cox, Willy Zwaenepoel - Scaling and availability for dynamic content web sites - *Rice University Technical Report TR02-395*, 2002.

[2] Christiana Amza, Alan L. Cox, Willy Zwaenepoel - Conflict-Aware Scheduling for Dynamic Content Applications - *Proceedings of USITS 2003*, March 2003

[3] Bela Ban - Design and Implementation of a Reliable Group Communication Toolkit for Java - Cornell University, September 1998.

[4] P. A. Bernstein, V. Hadzilacos and N. Goodman - Concurrency Control and Recovery in Database Systems - *Addison-Wesley*, 1987.

[5] Emmanuel Cecchet, Julie Marguerite and Willy Zwaenepoel - RAIDb: Redundant Array of Inexpensive Databases - *INRIA Research Report 4921* - September 2003.

[6] Emmanuel Cecchet, Julie Marguerite and Willy Zwaenepoel - C-JDBC: Flexible Database Clustering Middleware - *USENIX Annual Technical Conference 2004, Freenix track*, June 2004.

[7] P. Chen, E. Lee, G. Gibson, R. Katz and D. Patterson - RAID: High-Performance, Reliable Secondary Storage - *ACM Computing Survey*, volume 26, n2, pp. 145-185, 1994.

[8] Bettina Kemme and Gustavo Alonso - Don't be lazy, be consistent: Postgres-R, a new way to implement Database Replication - *Proceedings of the 26th International Conference on Very Large Databases*, September 2000.

[9] Bettina Kemme and Gustavo Alonso - A new approach to developing and implementing eager database replication protocols - *ACM Transactions on Database Systems* 25(3), pp. 333-379, 2000.

[10] Jim Gray, Pat Helland, Patrick O'Neil and Dennis Shasha - The Dangers of Replication and a Solution - *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, June 1996.

[11] Enhydra Octopus - http://octopus.enhydra.org/.

[12] Oracle - Oracle9i Real Application Clusters - *Oracle white paper*, February 2002.

[13] Mikael Ronstrom and Lars Thalmann - MySQL Cluster Architecture Overview - *MySQL Technical White Paper*, Avril 2004.

[14] D. Stacey - Replication: DB2, Oracle or Sybase - In *Database Programming & Design*, , 7(12), 1994.

[15] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme and G. Alonso - Database replication techniques: a three parameter classification - *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS2000)*, October 2000.

Ganymed: Scalable and Flexible Replication

Christian Plattner Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology (ETHZ)
ETH Zentrum, CH-8092 Zürich, Switzerland
{plattner, alonso}@inf.ethz.ch

Abstract

Transactional web applications play an important role in e-commerce environments and lead to significant scalability challenges to existing database architectures. Replication is an often proposed solution. Existing systems, however, often cannot guarantee consistency, scalability while supporting arbitrary access patterns. Ganymed, a middleware based replication solution, is intended to add scalability and adaptability to existing databases without having to give up consistency or having to change client software. By using a simple scheduling algorithm, RSI-GPC, the overall system footprint is small and therefore very efficient and easily extensible.

1 Introduction

Recent research in the area of replication for transactional web applications is mostly based either on middleware approaches [1, 2, 3] or on techniques that cache data at the edges of the network [4, 5, 6]. The use of caching can greatly reduce response times for large setups, however, strong consistency has to be given up. Middleware centered solutions that schedule requests over sets of replicas in turn can offer the same degree of consistency as single instance databases. Unfortunately, such systems suffer from at least one of the following problems: client software has to be modified to be able to work with the middleware (e.g., transaction access patterns have to be predeclared), efficient scheduling is only possible if the data is partitioned statically (e.g. leading to limited query patterns), certain features of the replicas can not be used (e.g., triggers are not supported), database logic is duplicated at the middleware level (e.g., conflict resolution, leading to limited scale-out due to table level locking). Middleware solutions often become the bottleneck of the system due to their complexity.

Our approach, Ganymed, is also a middleware based approach. However, we avoid all of those problem. Ganymed is tailored to the needs of typical transactional web applications. It guarantees full consistency while at the same time offering good scale out. There is no need to change client software, our current implementation features a JDBC driver that can be easily plugged into existing applications. The middleware scheduler, using our novel RSI-GPC algorithm, is able to balance transaction to a set of replicas, which can eben be different DBMS's. The data does not need to be partitioned, the system uses a fully replicated shared-nothing architecture. Update transactions that fire triggers do not impose a problem. The scheduler also does not involve SQL statement parsing or table level locking, leading to an efficient system with a small footprint.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Our TPC-W trace based evaluations show that Ganymed offers almost linear scalability for typical e-commerce workloads. Due to the graceful degradation properties of Ganymed, the availability of a database system can also be significantly improved.

2 The Ganymed Approach

Ganymed is a system which is able to execute transactions over a set of database replicas. There are two key ideas in Ganymed: The first is the separation of update and read-only transactions. Updates are handled by a master replica, queries are processed on a set of slave replicas. The second idea is to use snapshot isolation for the read-only transactions. Snapshot Isolation (SI) [7, 8] is a multiversion concurrency control mechanism used in databases. Popular database engines that use SI include Oracle [9] and PostgreSQL [10]. SI has the important property that readers are never blocked by writers, it is ideal for separating updates and queries. In Ganymed, all update transactions on the master result in writesets which have to be applied on the slave replicas. The use of snapshot isolation makes it then possible to install a stream of writesets from the master to the slaves without having conflicts with the read-only transactions on the slaves. This property of SI is a big win in comparison to systems that use traditional *two phase locking* (2PL), where many non-conflicting updates may be blocked by as simply as one reader.

However, Ganymed is not limited to database backends based on snapshot isolation. Whereas the slaves must implement snapshot isolation, this is not necessary for the master. Actually, the master replica can use any concurrency protocol that ensures that transactions do not see uncommitted data. The only requirement for a master is the ability to extract writesets of update transactions. As our evaluations have shown, this is achievable for popular database products like DB2, Oracle or PostgreSQL.

2.1 The RSI-GPC Algorithm

RSI-GC is short for *Replicated Snapshot Isolation with General Primary Copy*. A RSI-GPC based scheduler is responsible for a set of n fully replicated backend databases. One of the replicas is used as $master$, the other $n-1$ replicas are the $slaves$. The slaves must always implement snapshot isolation. The scheduler makes also a clear distinction between *read-only* and *update* transactions, which have to be marked by the client application in advance.

Update transactions: SQL statements of any arriving update transaction are directly forwarded to the master, they are never delayed. The scheduler takes notice of the order in which update transactions commit on the master. After a successful commit of an update transaction, the scheduler makes sure that the corresponding writeset is sent to the $n-1$ slaves and that every slave applies the writesets of different transactions in the same order as the corresponding commit occurred on the master replica. The scheduler uses also a global database version number. Whenever an update transaction commits on the master, the global database version number is increased by one and the client gets notified about the commit. Writesets get tagged by the version number which was created by the corresponding update transaction.

Read-Only Transactions: read-only transactions can be processed by any slave replica mode. The scheduler is free to decide on which replica to execute such a transaction. If on a chosen replica the latest produced global database version is not yet available, the scheduler must delay the creation of the read-only snapshot until all needed writesets have been applied to the replica. For clients that are not willing to accept any delays for read-only transactions there are two choices: either their queries are sent to the master replica, therefore reducing the available capacity for updates, or the client can set a staleness threshold. A staleness threshold is for example a maximum age of the requested snapshot in seconds or the condition that the client sees its own updates. The scheduler can then use this threshold to choose a replica. Once the scheduler has chosen a replica and the replica created a snapshot, all consecutive operations of the read-only transaction will be performed using that snapshot. Due to the nature of SI, the application of further writesets on this replica will not conflict with this or any other running read-only transaction.

Scheduler Performance: Due to its simplicity, there is no risk of a RSI-GPC scheduler becoming the bottleneck in the system. In contrast to other middleware based schedulers, like the ones used in [3, 1], this scheduling algorithm does not involve any SQL statement parsing or concurrency control operations. Also, no row or table level locking is done at the scheduler level. The detection of conflicts, which can only happen during updates, is left to the master replica. Moreover, unlike [3, 11], RSI-GPC does not make any assumptions about the data partition, organization of the schema, or answerable and unanswerable queries.

Fault Tolerance: Since only a small amount of state information must be kept by a RSI-GPC scheduler, it is even possible to construct parallel working schedulers. This helps to improve the overall fault tolerance. In contrast to traditional eager update-everywhere systems, where every replica has its own scheduler that is aware of the global state, the exchange of status information between a small number of RSI-GPC schedulers can be done very efficiently. Even in the case that all schedulers fail, it is possible to reconstruct the overall database state: a replacement scheduler can be used and its state initialized by inspecting all available replicas. In the case of a failing slave replica, the scheduler simply ignores it until it has been repaired by an administrator. However, in the case of a failing master, things are a little bit more complicated. By just electing a new master the problem is only halfway solved. First, the new master replica must feature the same concurrency control protocols as the old one for a seamless changeover. Second, the scheduler must also make sure that no updates from committed transactions get lost, thereby guaranteeing ACID durability. This goal can be achieved by sending commit notifications to clients after the writesets of update transactions have successfully been applied on a certain, user defined amount of replicas.

3 Prototype Architecture

The Ganymed system is already a full working system. Its main component is a lightweight middleware scheduler that implements the RSI-GPC algorithm to balance transactions from clients over a set of database replicas. Clients are typically application servers in e-commerce environments. From the viewpoint of such clients, the Ganymed scheduler behaves like a single database which offers SI for read-only transactions and (depending on the master replica) a set of concurrency control options for update transactions. Our working system prototype is entirely based on Java.

Figure 1 shows the main components of the architecture. Applications servers connect to the Ganymed scheduler through a custom JDBC driver. The Ganymed scheduler then distributes incoming transactions over the master and slave replicas. Writesets, denoted as WS, are extracted from the master and sent to the slaves. Replicas can be added and removed from the system at runtime. The master role can be assigned dynamically, for example when the master replica fails. The current prototype does not support parallel working schedulers, yet it is not vulnerable to failures of the scheduler. If a Ganymed scheduler fails, it will immediately be replaced by a standby scheduler. The decision for a scheduler to be replaced by a backup has to be made by the *manager component*. The manager component, running on a dedicated machine, constantly monitors the system. The manager component is also responsible for reconfigurations. It is used, e.g., by the database administrator to add and remove replicas. Interaction with the manager component takes place through a graphical interface.

Client Interface: Clients connect to the scheduler through the Ganymed JDBC 3.0 database driver. The availability of such a standard database interface makes it straightforward to connect Java based application

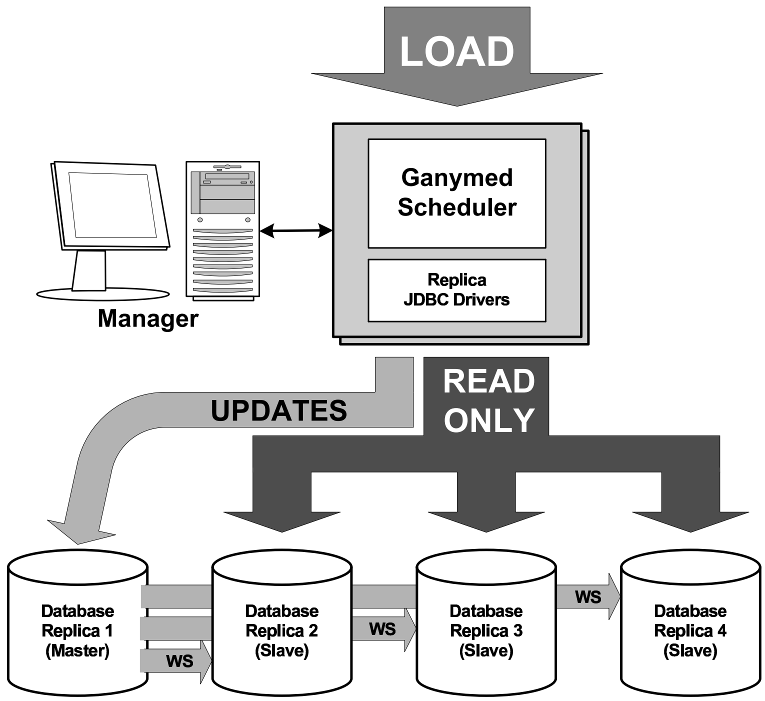

Figure 1: Ganymed Overview.

servers to Ganymed. The migration from a centralized database to a Ganymed environment is very simple, only the JDBC driver component in the application server has to be reconfigured, there is no change in the application code. Our driver also supports a certain level of fault tolerance. If a configured Ganymed scheduler is not reachable, the driver automatically tries to connect to an alternate scheduler.

Since the scheduler needs to know if a transaction is an update or read-only, the application code has to communicate this to the Ganymed JDBC driver. This mechanism is already included in the JDBC standard. Application code that wants to start a read-only transaction simply calls the *Connection.setReadonly()* method.

Writeset Extraction and Replica Support: On the replica side, Ganymed currently supports PostgreSQL and Oracle data-base engines. This allows to build heterogenous setups, where replicas run different database engines. Also, our approach is not limited to a specific operating system. As in the client side, the communication with the replicas is done through JDBC drivers. In a heterogenous configuration, the Ganymed scheduler has therefore to load for every different type of replica the corresponding JDBC driver. Since this can be done dynamically at run time, on startup the scheduler does not need to be aware of the type of replicas added at runtime.

Unfortunately, the JDBC interface has no support for writeset extraction, which is needed on the master replica. In the case of PostgreSQL, we implemented an extension of the database software. PostgreSQL is very flexible, it supports the loading of additional functionality during runtime. Our extension consists of a shared library written in C which holds the necessary logic to collect changes of update transactions in the database. Internally, the tracking of updates is done using triggers. To avoid another interface especially for the writeset handling, the extension was designed to be controlled over the normal JDBC interface. For instance, the extraction of a writeset can be performed with a ”*SELECT writeset()*” SQL query. The extracted writesets are table row based, they do not contain full disk blocks. This ensures that they can be applied on a replica which uses another low level disk block layout than the master replica.

In the case of Oracle, we implemented a similar extension, which is based on triggers and the JVM (Java

Virtual Machine) which is part of the Oracle database. As in the case of PostgreSQL, this extension is accessible through standard JDBC calls. Currently we are also implementing the writeset extraction feature for the IBM DB2 databases.

Implementation of RSI-GPC: Our current implementation of RSI-GPC is very strict, the scheduler always provides strong consistency. Loose consistency models are not supported in the current version, therefore read-only transactions will always see the latest snapshot of the database. The scheduler also makes a strict distinction between the master and the slave replicas. Even if there is free capacity on the master, read-only transactions are always assigned to a snapshot isolation based slave replica. This ensures that the master is not loaded by complex read-only transactions and that there is always enough capacity on the master for sudden bursts of updates. However, if there is no slave replica present, the scheduler is forced to assign all transactions to the master replica, acting as a relay.

Read-only transactions are assigned to a valid replica according to the LPRF (*least pending requests first*) rule. Valid means in this context, that the replica must contain the latest produced writeset. If no such replica exists, the start of the transaction is delayed. Once a transaction is assigned to a replica, be it of type update or a read-only, this assignment will not change. Also, unlike other replication solutions, Ganymed does not require that transactions are submitted as a block, i.e., the entire transaction must be present for it to be scheduled. In Ganymed different SQL statements of the same transaction are progressively scheduled as they arrive, with the scheduler ensuring that the result is always consistent.

To achieve a consistent state between all replicas, the scheduler must make sure that writesets of update transactions get applied on all replicas in the same order. This already imposes a problem when writesets are generated on the master, since the scheduler must be sure about the correct commit order of transactions. Ganymed solves this problem by sending COMMIT operations to the master in a serialized fashion. The distribution of writesets is handled by having a FIFO update queue for every replica. There is also for every replica a thread in the scheduler software that applies constantly the contents of that queue to its assigned replica.

4 Behavior for Typical TWA Loads

To be able to verify the validity of our approach we have performed extensive experiments with Ganymed. To ease the interpretation of the results, a homogenous setup was chosen: all the used replicas used the same database product, namely PostgreSQL, and all participating machines had the same hardware configuration. All replicas were initialized with the data set from a database of a TPC-W installation (scale factor: 10.000 items, 288.000 customers). The TPC benchmark W (TPC-W) is a transactional web benchmark from the Transaction Processing Council [12]. TPC-W defines an internet commerce environment that resembles real world, business oriented, transactional web applications. The benchmark also defines different types of workloads which are intended to stress different components in such applications. The TPC-W workloads are as follows: primarily *shopping*, *browsing* and web-based *ordering*. The difference between the different workloads is the ratio of browse to buy: browsing consists of 95% read-only interactions, for shopping the ratio is 80% and for ordering the ratio is 50%. Shopping, being the primary workload, is considered the most representative one. To be able to perform repeatable experiments with Ganymed, we generated a tracefile for each of the three workloads using a TPC-W installation.

The system was set up in different configurations and loaded with those traces. On one hand, the achievable throughput and the resulting response times were measured and compared to a single PostgreSQL instance. To do this, a load generator was attached to the database (either the single instance database or the scheduler, depending on the experiment). During a measurement interval of 100 seconds, a trace was then fed into the system over 100 parallel client connections and at the same time average throughput and response times were

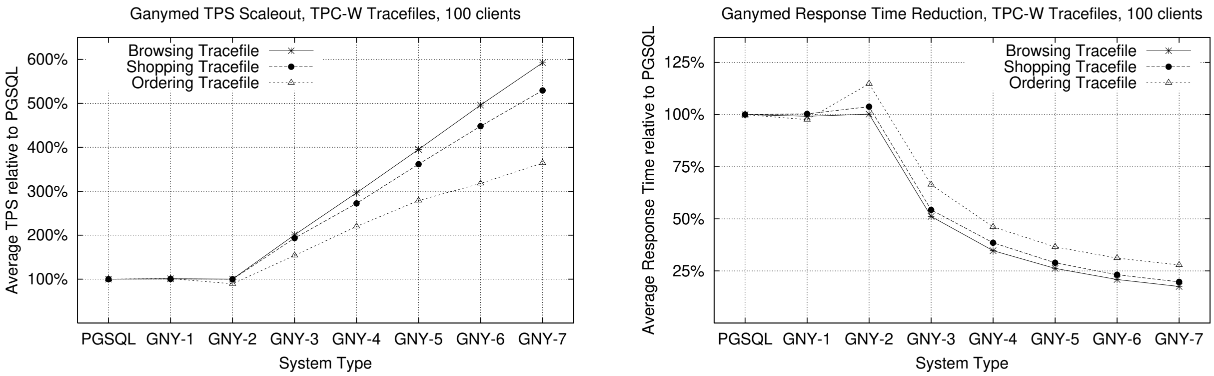

Figure 2: Ganymed Scaleout for TPC-W Tracefiles.

measured. On the other hand, the reaction of the system to failures was investigated. For this, the load generator was attached to the scheduler and then the change in throughput was monitored while disconnecting replicas.

Scalability: We measured the performance of the Ganymed scheduler in different configurations, from 1 up to 7 replicas. Including the single instance measurements, this results in a total of 8 experimental setups (called PGSQL and GNY-n, $1 \leq n \leq 7$). Figure 2 shows the results for the achieved throughput (transactions per second) and average transaction response times for the three TPC-W traces. The rate of aborted transactions was below 0.5 percent. As already noted, the TPC-W shopping workload is regarded as the most representative one. Clearly, a nearly linear scale-out in terms of throughput was achieved and response times were reduced. Obviously, in case of the ordering workload the system scales, but not linearly. This is due to the large amount of updates in this workload which have to handled by the master replica.

Fault Tolerance: As an example of the fault tolerance of Ganymed, we denote the results from an experiment were the master replica fails. The setup is a GNY-4 system, fed with a TPC-W shopping trace. During the measurements, a SIGKILL signal was used to stop the PostgreSQL database system software running on the master replica. The scheduler reacted by reassigning the master role to a different, still working slave replica. It is important to note that the reaction to failing replicas can be done by the scheduler without intervention from the manager console. Even with a failed or otherwise unavailable manager console the scheduler can still disable failed replicas and, if needed, move the master role autonomously. Figure 3 shows the resulting throughput histogram for this experiment. Transaction processing is normal until in second 45 the master replica stops working. The immediately move of the master role to a slave replica leaves a GNY-3 configuration with one master and two slave replicas. The failure of the master replica led to an abort of 2 update transactions, no other transactions were aborted during the experiment. The arrows in the graph show the change of the average transaction throughput per second.

5 Conclusions

Despite its simplicity, the Ganymed system based on RSI-GPC tries to avoid common drawbacks of existing solutions. For instance, in comparison to other middleware based approaches like [1, 2, 3], Ganymed does not involve any SQL statement parsing or table level locking. Due to the use of SI and the separation of updates and read-only transactions, conflicts can only happen at the master replica. Therefore, this task has not to be duplicated at the scheduler level and can be done efficiently by the master replica. SQL parsing and table level

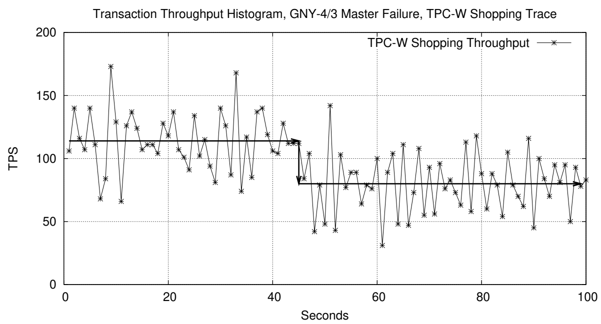

Figure 3: Ganymed Fault Tolerance.

locking at the scheduler both severely limit concurrency and prevent the use of certain features in the database replicas. Triggers, for example, cannot be used with such approaches: the scheduler cannot easily decide by looking at SQL statements if they fire triggers (and eventually update other tables), the only solution would be to lock even more conservatively. In the Ganymed approach, triggers do not impose a problem. They can be installed and changed on the master replica at any time and the scheduler does not have to be informed about their existence.

To circumvent these problems, some systems force the client to predeclare all the objects a transactions is going to access. However, this is very inconvenient and inflexible. It also complicates the adaptation of already existing client software. Solutions where the client has to send full transactions as a block impose similar, even worse problems.

Another problem of common middle schedulers solutions is the usage of write-all approaches on SQL statement level: updates are applied to replicas by broadcasting the client's SQL update statements. This can be very inefficient, since SQL processing work is duplicated at all replicas. In the case of complex update statements which update just a few rows, it is much more efficient to work with writesets. In the case of Ganymed, SQL update statements have only to be evaluated at the master; slave replicas just apply the resulting changes, leaving capacity for read-only transactions.

Systems that are distributed over the network and support caching at the edge level, like [4, 5, 6], can dramatically improve response times. Also, part of these systems are very flexible and can react to changes in the load and adapt the used cache structures. Unfortunately, in all cases full consistency has to be given up to achieve scalability.

The goal of Ganymed is to be as flexible as possible without having to give up full consistency or disallowing certain access patterns. Ganymed imposes no data organization, structuring of the load, or particular arrangements of the schema. Applications that want to make use of Ganymed do not have to be modified, the JDBC driver approach guarantees the generality of the interface that Ganymed offers. Thanks to the minimal infrastructure needed, Ganymed provides excellent scalability and reliability for typical transaction web applications. For typical loads, Ganymed scales almost linearly. Even if replicas fail, Ganymed is able to continue working with proper performance levels thanks to the simple mechanisms involved in recovery.

As part of future work, we are exploring the use of specialized indexes in the read-only replicas to speed up query processing. We are also studying the possibility of autonomic behavior in the creation of such indexes or even whole replicas. For instance, the manager console could adapt dynamically the replica setup as a result of inspecting the current load. Given the low overhead of the infrastructure, we can invest in such optimizations without worrying about the impact of the extra computations on performance. In the medium term, the implementation of complete autonomous behavior is an important goal.

References

[1] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Distributed Versioning: Consistent Replication for Scaling Back-End Databases of Dynamic Content Web Sites. In *Middleware 2003, ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, June 16-20, 2003, Proceedings*, 2003.

[2] Cristiana Amza, Alan L. Cox, and Willy Zwaenepoel. Conflict-Aware Scheduling for Dynamic Content Applications. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[3] Emmanuel Cecchet, Julie Marguerite, Mathieu Peltier, and Nicolas Modrzyk. C-JDBC: Clustered JDBC, http://c-jdbc.objectweb.org.

[4] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache Tables: Paving the Way for an Adaptive Database Cache. In *Proceedings of the 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany*, 2003.

[5] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A Dynamic Data Cache for Web Applications. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, 2003.

[6] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. Transparent Mid-tier Database Caching in SQL Server. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pages 661–661. ACM Press, 2003.

[7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 1–10, May 1995.

[8] Ralf Schenkel and Gerhard Weikum. Integrating Snapshot Isolation into Transactional Federation. In Opher Etzion and Peter Scheuermann, editors, *Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel, September 6-8, 2000, Proceedings*, volume 1901 of *Lecture Notes in Computer Science*, pages 90–101. Springer, 2000.

[9] Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7. Oracle White Paper, July 1995.

[10] PostgreSQL Global Development Group. PostgreSQL: The most advanced Open Source Database System in the World. http://www.postgresql.org.

[11] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *IEEE 22nd Int. Conf. on Distributed Computing Systems, ICDCS'02, Vienna, Austria*, pages 477–484, July 2002.

[12] The Transaction Processing Performance Council. TPC-W, a Transactional Web E-Commerce Benchmark. http://www.tpc.org/tpcw/.

MTCache: Mid-Tier Database Caching for SQL Server

Per-Åke Larson Jonathan Goldstein
Microsoft
{palarson,jongold}@microsoft.com

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

Jingren Zhou
Columbia University
jrzhou@cs.columbia.edu

Abstract

MTCache is a prototype mid-tier database caching solution for SQL Server that transparently offloads part of the query workload from a backend server to front-end servers. The goal is to improve system throughput and scalability but without requiring application changes. This paper outlines the architecture of MTCache and highlights several of its key features: modeling of data as materialized views, integration with query optimization, and support for queries with explicit data currency and consistency requirements.

1 Introduction

Many applications today are designed for a multi-tier environment typically consisting of browser-based clients, mid-tier application servers and a backend database server. A single user request may generate many queries against the backend database. The overall response time seen by a user is often dominated by the aggregate query response time, particularly when the backend system is highly loaded. The goal of mid-tier database caching is to transfer some of the load from the backend database server to front-end database servers. A front-end server replicates some of the data from the backend database, which allows some queries to be computed locally.

A key requirement of mid-tier database caching is application transparency, that is, applications should not be aware of what is cached and should not be responsible for routing requests to the appropriate server. If they are, the caching strategy cannot be changed without changing applications.

This paper gives a brief overview of MTCache, a mid-tier database cache solution for Microsoft SQL Server that achieves this goal. We outline the architecture of our prototype system and highlight a few specific features:

- Modeling of local data as materialized views, including a new type called partially materialized views.
- Integration into query optimization and improvements for parameterized queries
- Support for explicit currency and consistency requirements

The rest of the paper is organized as follows. Section 2 outlines the overall architecture of MTCache. Section 3 deals with query processing and optimization and describes optimizer extensions for parameterized queries. Section 4 describes the support for explicit currency and consistency requirements and section 5 introduces partially materialized views. More details can be found in references [6] and [7].

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

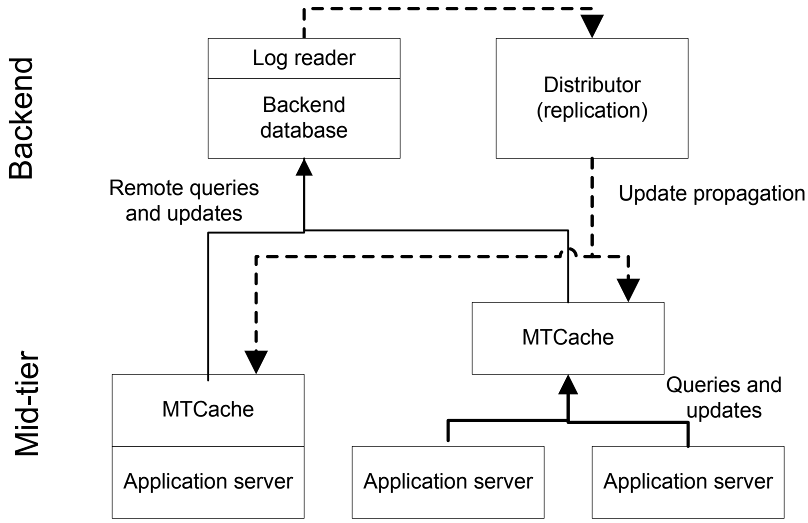

Figure 1: Configuration with two MTCaches and three application servers

2 MTCache architecture

Figure 1 illustrates a configuration with two MTCache servers. An MTCache server can run either on the same machine as an application server or on a separate machine, possibly handling queries from several application servers.

An MTCache front-end server stores a 'shadow' database containing exactly the same tables, views, indexes, constraints, and permissions as the backend database but all the shadow tables, indexes and materialized views are empty. However, all statistics on the shadow tables, indexes and materialized views reflect their state on the backend database. Shadowing the backend catalog information on the caching server makes it possible to locally parse queries, perform view matching and check permissions. The shadowed statistics are needed during query optimization.

The actual data stored in an MTCache database is a subset of the data from the backend database. The subset consists of fully or partially materialized select-project views of tables or materialized views residing on the backend server. (Partially materialized views are explained in section 5.) What data to cache is controlled by the DBA who simply creates a collection of materialized views on the MTCache server.

The cached views are kept up to date by replication. When a cached view is created on the mid-tier server, we automatically create a replication subscription (and publication if needed) and submit it. Replication then immediately populates the cached view and begins collecting and forwarding applicable changes. The cached data can be thought of as a collection of distributed materialized views that are transactionally consistent but may be slightly out of date.

An MTCache server may cache data from multiple backend servers. Each shadow database is associated with a particular backend server but nothing prevents different databases on a cache server from being associated with different backend servers. The same applies to replication: different databases may receive data from different distributors.

All queries are submitted to the MTCache server whose optimizer decides whether to compute a query locally, remotely or part locally and part remotely. Optimization is entirely cost based. All inserts, deletes and updates are submitted to the MTCache server, which immediately forwards them to the backend server. Changes are then propagated asynchronously to all affected MTCache servers.

Applications connect to a database server by connecting to an ODBC source. An ODBC source definition maps a logical name to an actual server instance and specifies a number of connection properties. To cause an application to connect to the front-end server instead of the backend server, we only need to redirect the appropriate ODBC source from the backend server to the front-end server.

3 Query optimization and processing

Queries go through normal optimization on the MTCache server. Cached views are treated as regular materialized views and, if applicable, picked up by the optimizers view matching mechanism [4]. However, even if all the necessary data is available locally, the query is not necessarily evaluated locally. It may be faster to evaluate the query on the backend if, for example, the backend has an index that greatly speeds up processing. The reverse is also true; even if none of the required data is available locally, it may be worthwhile evaluating part of the plan locally. An extreme example is a query that computes the Cartesian product of two tables. It is cheaper to ship the individual tables to the local server and evaluate the join locally than performing the join remotely and shipping the much larger join result. The optimizer makes a cost-based decision on whether to evaluate a subexpression locally or remotely.

To add this capability to the optimizer, we introduced a new operator, DataTransfer, and a new physical property, DataLocation, for each data source and operator. DataLocation can be either Local or Remote. The DataTransfer operation simply changes the DataLocation property from Remote to Local or vice versa. All other operators leave DataLocation unchanged. Cached views and their indexes are Local and all other data sources are Remote. A new optimization rule adds a DataTransfer operation whenever the parent requests a Local result and the input expression is Remote. DataTransfer is a (physical) property enforcer, similar to sorting. The estimated cost of a DataTransfer operation is proportional to the estimated volume of data shipped plus a startup cost.

Cost estimation was modified to favor local execution over execution on the backend server. All cost estimates of remote operations are multiplied by a small factor (greater than 1.0). The motivation is that, even though the backend server may be powerful, it is likely to be heavily loaded so we will only get a fraction of its capacity.

The final plan produced by the optimizer may be a completely local plan, a completely remote plan or a combination thereof. Subexpressions to be evaluated remotely are easy to find in the plan: simply look for DataTransfer operators. Every subexpression rooted by a DataTransfer operator is converted to a (textual) SQL query and sent to the backend server during execution.

Parameterized queries require special care if we want to make maximal use of the cached data. Suppose we have a cached selection view, Cust1000 containing all customers with cid <= 1000 and receive the query

```
select cid, cname, caddress
from customer where cid <= @p1
```

where @p1 is a run-time parameter. The view cannot always be used; only when the actual value of @p1 is less than or equal to 1000. Unfortunately, the actual parameter value is only known at run time, not at optimization time. To fully exploit views like Cust1000 also for parameterized queries, the optimizer generates plans with two branches, one local branch and one remote branch, with a SwitchUnion operator on top that selects the appropriate branch at run time. During view matching, it is noticed that Cust1000 contains all required rows if @p1 <= 1000 but not otherwise. The optimizer then creates an alternative with a SwitchUnion operator on top having the switch predicate @p1 <= 1000 and two children, one using the cached view Cust1000 and the other submitting a query to the backend server. During execution, the switch predicate is evaluated first and the local branch is chosen if it evaluates to true, otherwise the remote branch.

4 Currency and consistency requirements

Many applications routinely make use of cached or replicated data that may be somewhat stale, that is, the data does not necessarily reflect the current state of the database. However, current database system do not allow an application to indicate what level of staleness is acceptable. We have extended SQL so queries can include explicit data currency and consistency (C&C) requirements and MTCache guarantees that the query result satisfies the requirements. Explicit C&C requirements give the DBMS freedom to decide on caching and other strategies to improve performance while guaranteeing that applications requirements are still met.

Let us first clarify what we mean by the terms currency and consistency. Currency simply refers to how current or up-to-date a set of rows (a table, a view or a query result) is. Consider a database with two tables, Books and Reviews, as might be used by a small online book store. Suppose that we have a replicated table BooksCopy that is refreshed once every hour. The currency of BooksCopy is simply the elapsed time since the last refresh to the commit time of the latest transaction updating Books 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 two replicas do not necessarily reflect exactly the same snapshot. If they do, we say that they are (mutually) consistent.

C&C constraints are expressed through a optional currency clause that occurs last in a Select-From-Where (SFW) block and follows the same scoping rules as the WHERE clause. We will illustrate what can be expressed through a few example queries.

```
Q1: Select ...
     from Books B, Reviews R
     where B.isbn = R.isbn and B.price < 25
     currency bound 10 min on (B,R)
```

The currency clause in Q1 expresses two constraints: a) the inputs cannot be more than 10 min out of date and b) the two inputs must be consistent, that is, be from the same database snapshot. Suppose that we have cached replicas of Books and Reviews, and compute the query from the replicas. To satisfy the C&C constraint of Q1, the result obtained using the replicas must be equivalent to the result that would be obtained if the query were computed against mutually consistent snapshots of Books and Reviews that are no older than 10 min (when execution of the query begins).

```
Q1: Select ...
     from Books B, Reviews R
     where B.isbn = R.isbn and B.price < 25
     currency bound 10 min on B, 30 min on R
```

Query Q2 relaxes the bound on R to 30 min and no longer requires that the inputs be mutually consistent. For a catalog browsing application, for example, this level of currency and consistency may be perfectly acceptable.

Q1 and Q2 required that all input rows be from the same snapshot, which may be stricter than necessary. Sometimes it is acceptable if rows or groups of rows from the same table are from different snapshots, which is illustrated by Q3. 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.

```
Q3: Select ...
     from Reviews R
     currency bound 10 min on R by R.isbn
```

MTCache enforces C&C constraints when specified. Consistency constraints are enforced at optimization time while currency constraints are enforced at execution time. MTCache keeps track of which cached views are guaranteed to be mutually consistent and how current their data is. We extended the SQL Server optimizer to select the best plan taking into account the query's C&C constraints and the status of applicable cached views. In contrast with traditional plans, the execution 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. More details can be found in reference [7].

5 Partially materialized views

In current database systems, a view must be either fully materialized or not materialized at all. Sometimes it would be preferable to materialize only some of the rows, for example, the most frequently accessed rows, and be able to easily and quickly change which rows are materialized. This is possible in MTCache by means of a new view type called a partially materialized view (PMV). Exactly which rows are currently materialized is defined by one or more control tables (tables of content) associated with the view. Changing which rows are materialized is a simple matter of updating the control table.

We will illustrate the idea using the Books and Reviews database mentioned earlier. Suppose we receive many queries looking for paperbacks by some given author and follow-up queries looking for related reviews. Some authors are much more popular than others and which authors are popular changes over time. In other words, the access pattern is highly skewed and changes over time. We would like to answer a large fraction of the queries locally but it is too expensive in storage and maintenance required to cache the complete Books and Reviews tables. Instead, we cache only the paperbacks of the most popular authors and the associated reviews. This scenario can be handled by creating one control table and two partially materialized views as shown below.

```
Create table AuthorList( authorid int)

Create view BooksPmv as
  select isbn, title, price, ...
  from Books
  where type = 'paperback' and authorid in (select authorid from AuthorList)

Create view ReviewsPmv as
  select isbn, reviewer, date, ...
  from Reviews
  where isbn in (select isbn from BooksPmv)
```

AuthorList act as a control table and contains the Ids of the authors whose paperback books are currently included in the view BooksPmv. The contents of BooksPmv is tied to the entries of AuthorList by the subquery predicate in the view definition. BooksPmv in turn acts as a control table for ReviewsPmv because it includes reviews only for books that are found in BooksPmv. In this way the contents of the two views are coordinated.

To add information about a new author to the two views, all that is needed is to add the author's id to AuthorList. Normal incremental view maintenance will then automatically add the author's paperback books to BooksPmv, which in turn causes the related reviews to be added to ReviewsPmv.

View matching has been extended to partially materialized views and also produces dynamic plans. Suppose we receive a query looking for paperbacks by an author with authorid=1234. This query can potentially be answered from BooksPmv but it is not guaranteed because its content may change at any time. To handle this situation, the optimizer produces a dynamic plan with a SwitchUnion on top that selects between using

BooksPmv and a fallback plan that retrieves the data from the backend server. Determining which branch to use consists of checking whether authorid=1234 exists in AuthorList.

This brief description covers only the simplest form of PMVs, namely, PMVs with single discrete-value control tables. Other types of control tables are possible, for example, range control tables and a PMV may have several control tables. Further details will be provided in an upcoming paper.

6 Related Work

Our approach is similar to DBCache [8, 1, 2, 3] in the sense that both systems transparently offload some queries to front-end servers, forward all updates to the backend server and rely on replication to propagate updates. DBCache was originally limited to caching complete tables [8] but more recently has added support for dynamically determined subsets of rows by means of a new table type called Cache Tables [3]. MTCache provides additional flexibility by modeling the cache contents as fully or partially materialized views, thereby allowing caching of horizontal and vertical subsets of tables and materialized views on the backend. DBCache appears to always use the cached version of a table when it is referenced in a query, regardless of the cost. In MTCache this is not always the case: the decision is fully integrated into the optimization process and is entirely cost-based.

TimesTen also offers a mid-tier caching solution built on their in-memory database manager [9, 10]. Their product provides many features but the cache is not transparent to applications, that is, applications are responsible for query routing.

References

[1] M. Altinel, Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. G. Lindsay, H. Woo, L. Brown DB-Cache: Database Caching for Web Application Servers, SIGMOD 2002, 612.

[2] M. Altinel, C. Bornhvd, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. Reinwald: Cache Tables: Paving the Way for an Adaptive Database Cache. VLDB 2003, 718-729

[3] C. Bornhovd, M. Altinel, S. Krishnamurthy, C. Mohan, H. Pirahesh, B. Reinwald, DBCache: Middle-tier Database Caching for Highly Scalable e-Business Architectures, SIGMOD 2003, 662.

[4] J. Goldstein, P.-Å. Larson, Optimizing Queries Using Materialized Views: A practical, scalable solution. SIGMOD 2001, 331-342.

[5] P.-Å. Larson, J. Goldstein, J. Zhou, Transparent Mid-Tier Database Caching in SQL Server, SIGMOD 2003, 661.

[6] P.-Å Larson, J. Goldstein, J. Zhou, MTCache: Transparent Mid-Tier Database Caching in SQL Server. ICDE 2004, 177-189

[7] H. Guo, P.-Å Larson, R. Ramakrishnan, J. Goldstein, Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. SIGMOD 2004 (to appear).

[8] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, J. F. Naughton, Middle-Tier Database Caching for e-Business, SIGMOD 2002, 600-611.

[9] The TimesTen Team, High Performance and Scalability through Application-Tier, In-Memory Data Management, VLDB 2000, 677-680.

[10] The TimesTen Team, Mid-Tier Caching: The TimesTen Approach, SIGMOD 2002, 588-593.

ICDE 2005 TOKYO
International Conference on Data Engineering

The 21st International Conference on Data Engineering (ICDE 2005)

April 5-8, 2005
National Center of Sciences, Tokyo, Japan
Sponsored by
The IEEE Computer Society
The Database Society of Japan (DBSJ)
Information Processing Society of Japan (IPSJ) (pending final approval)
The Institute of Electronics, Information and Communication Engineers (IEICE) (pending final approval)
http://icde2005.is.tsukuba.ac.jp/ http://icde2005.naist.jp/ (mirror)

Call for Papers

Scope

Data Engineering deals with the use of engineering techniques and methodologies in the design, development and assessment of information systems for different computing platforms and application environments. The ICDE 2005 International Conference on Data Engineering provides a premier forum for:

- sharing research solutions to problems of today's information society
- exposing practicing engineers to evolving research, tools, and practices and providing them with an early opportunity to evaluate these
- raising awareness in the research community of the problems of practical applications of data engineering
- promoting the exchange of data engineering technologies and experience among researchers and practicing engineers
- identifying new issues and directions for future research and development work.

Welcome to Tokyo

The conference will be held in Tokyo, the capital city of Japan. Tokyo is also the most populous metropolitan area in the world. From its establishment in 1603 to the end of the 19th century, Tokyo prospered as a castle town called Edo. Tokyo still remains an exciting and attractive city that is constantly developing. Yet despite its complex urban landscape and impressive architecture, this city abounds with parks and gardens. Particularly in this season, blooming cherry blossoms will welcome you to Tokyo. Well-known sites close to Tokyo include the Great Buddha statue at Kamakura and scenic Mt. Fuji.

Topics of Interests

ICDE 2005 invites research submissions on all topics related to data engineering, including but not limited to those listed below:

- Database System Internals and Performance
- Query Processing and Optimization
- Data Warehouse, OLAP, and Statistical Databases
- Mining Data, Text and Web
- Semi-structured Data and XML
- Middleware, Web Services, and Workflow
- Heterogeneity, Semantic Web, and Metadata
- Privacy and Security
- Stream Processing, Continuous Queries, and Sensor Databases
- Database Applications and Experiences
- Temporal, Spatial, Scientific, and Biological Databases
- Distributed, Parallel, Deep Web, and P2P Databases
- Data Management for Ubiquitous and Mobile Computing

Conference Officers

Honorary Chair: The Late Yahiko Kambayashi (Kyoto University, Japan)

Organizing Committee Chair: Yoshifumi Masunaga (Ochanomizu University, Japan)

General Chairs: Rakesh Agrawal (IBM Almaden Research Center, USA)
Masaru Kitsuregawa (University of Tokyo, Japan)

Program Chairs: Karl Aberer (EPF Lausanne, Switzerland)
Michael Franklin (UC Berkeley, USA)
Shojiro Nishio (Osaka University, Japan)

Executive Committee Chair: Jun Adachi (National Institute of Informatics, Japan)

Panel Chairs: Umeshwar Dayal (HP Labs, USA)
Hongjun Lu (Hong Kong Univ. of Science & Technology, China)
Hans-Jörg Schek (UMIT, Austria, and ETH Zurich, Switzerland)

Tutorial Chairs: Michael J. Carey (BEA Systems, USA)
Stefano Ceri (Politecnico di Milano, Italy)
Kyu-Young Whang (KAIST, Korea)

Industrial Chairs: Anand Deshpande (Persistent Systems, India)
Anant Jhingran (IBM Silicon Valley Lab, USA)
Eric Simon (Medience, France)

Demo Chair: Ling Liu (Georgia Institute of Technology, USA)

Local Arrangement Chair: Haruo Yokota (Tokyo Institute of Technology, Japan)

Workshop Chair: Masatoshi Yoshikawa (Nagoya University, Japan)

Proceedings Chair: Motomichi Toyama (Keio University, Japan)

Publicity Chair: Hiroyuki Kitagawa (University of Tsukuba, Japan)

DBSJ Liaison: Hiroshi Ishikawa (Tokyo Metropolitan University, Japan)

Treasurer: Miyuki Nakano (University of Tokyo, Japan)

Program Committee Area Chairs

Anastassia Ailamaki (Carnegie Mellon University, USA)
Gustavo Alonso (ETH Zurich, Switzerland)
Phillip Gibbons (Intel Research, USA)
Takahiro Hara (Osaka University, Japan)
Jayant R. Haritsa (IISc Bangalore, India)
Alfons Kemper (University of Passau, Germany)
Sharad Mehrotra (UC Irvine, USA)
Wolfgang Nejdl (University of Hannover, Germany)
Jignesh M. Patel (University of Michigan, USA)
Evaggelia Pitoura (University of Ioannina, Greece)
Jayavel Shanmugasundaram (Cornell University, USA)
Kyuseok Shim (Seoul National University, Korea)
Kian-Lee Tan (National University of Singapore, Singapore)

Important Dates

Abstract deadline:	June 24(THU), 2004 2pm PST
Submission deadline:	July 1(THU), 2004 2pm PST
Notification date:	September 13(MON), 2004

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398