Examining the Performance of a Constraint-Based Database Cache

Andreas Bühmann and Joachim Klein

Databases and Information Systems, Department of Computer Science,
University of Kaiserslautern, P. O. Box 3049, 67653 Kaiserslautern, Germany
buehmann@informatik.uni-kl.de, jklein@informatik.uni-kl.de

Abstract. Constraint-based database caching aims at correctly answering SQL query predicates from a local cache database by exploiting constraints that have previously been used in selecting sets of records to be cached from a remote database.
In this paper, we take our first steps in looking at performance aspects of our prototype Adaptive Constraint-based Cache (ACCache), which is realized in a middleware manner on top of regular databases. We describe the measurement framework we have developed for analyzing distributed systems in general and our extensions for observing the ACCache system. Within our measurement setup, the initial focus is on two central ACCache functions: query processing and cache loading. To demonstrate their time behavior and interaction, we have chosen a scenario based on the TPC-W specification. We conclude with a discussion of our first measurement results.

1 Motivation

Applications that interact with real-world users typically strive for good (or at least acceptable) response times. This is a particular challenge if the application routinely relies on the services of a central backend database (DB) system that is located far from the application, e. g., in a Web scenario where application servers have been spread around the world at the "edge" of the Web to reduce their distance to the users. In this scenario with usually a large number of users, relieving the backend system of some of its load becomes equally important.

Caching is a means to approach these two aims: By intercepting requests to a remote system component and constructing responses locally (from earlier responses), communication costs to and processing costs on the remote component can be saved. Caching can be performed on various levels within an information-system infrastructure: For example, generated Web pages (or fragments thereof) can be cached, persistent objects within an application server, or pages of a database in a DB buffer.

Database caching is located at the level of logical data structures (such as tables and records in a relational DB) and higher query languages (such as SQL). The goal is to have a cache in the path from the application to the backend DB that is as transparent as possible and that is able to process SQL queries locally

based on locally stored parts of the backend DB. The constraint-based approach to database caching maintains a selection of cache tables, each containing a subset of records of the corresponding backend table. Cache constraints restrict what constitutes a valid state of the cache such that deciding what is in the cache and which predicates can be answered becomes easy.

The rest of this paper is structured as follows: In Sect. 2, we give an overview of how constraint-based database caching works and how it is implemented in our prototype system ACCache. We then turn to measurements in ACCache and describe our measurement framework and our concrete measurement setup in Sect. 3. The results of our first measurements are presented in Sect. 4, before we give an outlook on future extensions in Sect. 5.

2 Constraint-based Database Caching

In the general database-caching scenario, there are a backend (BE) database, which holds all data, and one or more cache databases, which contain varying subsets of that data. Ideally, the cache databases would contain data needed often in the nearer future.

With our model of constraint-based DB caching, *cache groups* are used to describe what data is to be kept in the cache and what constraints the cache contents have to fulfill at any time. These constraints can later be utilized to reason about whether a query can be (partly) answered from the cache.

For selected backend tables T_{B}, a cache group includes a corresponding cache table T with the same schema, i. e., for each column $T_{\mathrm{B}}.c$ in the backend table there is a column $T.c$ of same type (incl. unique constraints) in the cache table. (Foreign key constraints are not copied to the cache.)

2.1 Completeness and Constraints

For DB caching, *completeness* is a most important concept: Having all the records that are needed to evaluate a certain predicate in the cache is known under the term *predicate completeness* [1]. Completeness of more complex predicates is achieved by starting with completeness of very simple equality predicates and extending them with the help of cache constraints.

Equality predicates (EPs) of the type $T.c = v$, where v is a value of column $T.c$, are supported by the completeness of v. This value v is *complete* in a cache column $T.c$ if all records from T_{B} that have this value in c are in the cache (in T).

A *referential cache constraint (RCC)* is a value-based relationship between two columns: a source column $S.a$ and a target column $T.b$. An RCC $S.a \rightarrow T.b$ guarantees that every value in $S.a$ (in the cache!) is complete in $T.b$. This allows an equi-join (EJ) $S.a = T.b$ to be performed in the cache, once it has been verified that the needed S records (specified by other predicates such as $S.b = v$) are in the cache.

Basically, this procedure allows us to deal with predicates of the form $EP \wedge EJ_1 \wedge EJ_2 \wedge EJ_n$ in the cache, where all of the equi-joins EJ and the equality

predicate *EP* are connected via some tables. More complex predicates that can be constructed from this simple type by con-/disjunction and by further restrictions could also be processed in the cache.

2.2 Probing and Query Execution

When a query reaches the cache, it has to be decided whether the query can be answered partially in the cache and what part of the query result must be fetched from the backend. Deciding on the completeness of a (partial) predicate in the cache is done in two phases:

1. For each equality predicate $T.c = v$, which compares a column $T.c$ to a value v, completeness of v is decided by *probing* the cache.
2. Starting from complete values providing entry points for the query into the cache, RCCs matching equality predicates of type $S.a \rightarrow T.b$ in the query predicate are then used to extend the completeness to the largest predicate possible.

Probing works by issuing simple existence queries for values in some columns: You might know from prior analysis that all values in a cache column are complete (column completeness [1]), or you can leverage the RCCs by probing in their source columns. Either way, the existence of a value implies its completeness in a (possibly different) column.

Once the partial predicate that is complete in the cache has been found, it is clear that, for the tables referenced in that predicate, their cache counterparts can be used for executing the query. For the remaining tables, the original table at the backend must be accessed.

2.3 Loading and Unloading

Records are loaded into the cache whenever there is a hint that they will be needed in the future. *Filling columns* are responsible for providing these hints: As soon as specific value v of a filling column f is referenced in a query, v is made complete in the cache and subsequently fulfilling RCCs makes sure that a neighborhood of related records becomes available in the cache, too.

Loading is guided by the graph of RCCs: The sets of records to be inserted in the cache can be determined by following the RCCs in their natural direction: Usually, records inserted into the source table of an RCC demand matching records to be loaded into the target table. The actual insertion of those record sets into the cache tables may be performed in the reverse order (bottom-up) to provide more consistent cache states during the loading and thus better concurrency with readers [2].

Unloading aims at reversing the process of loading but has to cope with added difficulties due to records being required by multiple other records via multiple RCCs.

2.4 Prototype ACCache

Our prototype implementation of the techniques just sketched is called ACCache (Adaptive Constraint-based Cache) [2]. It employs a middleware strategy to realize the behavior of the database cache on top of two regular databases (backend and cache) that are accessed via JDBC: Probing, (un)loading, and maintenance of RCCs is done via (prepared) SQL statements. Query processing leverages the federated-query functionality of the underlying database management system to be able to access backend as well as cache tables within a single SQL query that is a rewrite of the original user query. (To the outside, ACCache implements a JDBC interface.)

Data to be unloaded from the cache is chosen based on access statistics, but the unloading itself is not performed yet. At our current stage, we start out with an empty cache and consider only a number of loading operations and their influence on query performance.

3 Measurement Setup

For performing measurements in the ACCache system, we use a framework developed in-house [3]. It supports a developer in setting up and executing measurements for a distributed system. The framework's components offer a wide range of functionality for measuring distributed structures. In detail, the framework supports:

- capturing the structure of the distributed system to represent all information needed for a measurement run
- a generic way to simulate a client node and its workload
- implementing a specific observer component in a simple way
- designing context objects for tracing the call structure on a component of the distributed system. For each context object the capturable measurement values are specified.
- asynchronous transmission of the measurement values transparently to the developer (implemented with the Java Messaging System)
- writing the measurement values to a database by using a default data writer
- automatic generation of the classes and interfaces needed to capture the measurement values and the database schema to save these values.

In addition, the framework supports automated execution of configured measurement runs if adequate automation steps have been implemented. The complexity of these automations is not bounded: They range from the automatic configuration of an application before the measurement starts up to the automatic installation of database systems or operating systems. To be able to return to the state prior to the measurement and to allow for an automated run of multiple measurements in succession, every automation step must be capable of undoing its configurations (if necessary).

In order to understand how the measurements for the constraint-based database caching were performed, we will first take a look at the course of a general measurement as dictated by the framework and then concentrate on some of its concepts in detail.

3.1 Working Nodes

A *working node* represents an application within the distributed system we want to measure. For each working node, it has to be defined how the node can be called or accessed within the distributed system. To reproduce the call structure of the system, each working node can be connected to suitable other ones.

A special working node in our framework is the *simulated client*, which can be used to simulate client operations. For this, it is possible to create a set of work units, which can be added to a work-unit scheduler, which is then used by a simulated client to perform the workload. The scheduler uses two parameters, *maxExecutions* and *overallPercent*, to decide which work unit is to be performed next. The parameter *overallPercent* describes how often a work unit should run in comparison with other work units added to the scheduler.

For our first measurement, we have built three working nodes: the *BackendWorkingNode* to represent the backend database system, the *ACCacheWorkingNode* for the ACCache system, and a simulated client called *JDBCClient*, which generates the workload during the measurement (see also Fig. 1).

3.2 Capturing Measurement Values

To capture values from an application represented by a working node, an observer needs to be defined. The observer has to register itself at the application it would like to observe. Hence, the application has to provide an appropriate observation interface.

We use *execution contexts* to determine the values which can be observed and the structure of dependencies between them. In addition, the interfaces of the execution contexts can be used to implement an observation interface if the application does not already have one (this only works if the observed application is written in Java and can be modified because its source code is available). An execution context is characterized by the capturable values and the creatable child execution contexts. For a given execution context, a particular value can be captured once (e. g., starting time) or multiple times (e. g., resource utilization over time).

This method of constructing execution contexts allows us to design the order and structure of events in an application as well as the observable measurement values. From this design, interfaces and classes can be generated that can be used by the observer to transmit measurement values to the measurement manager. The generated classes are also used to write received measurement values to a database.

For our ACCache measurements, our overall setup including the measured components (square boxes) as well as the measuring components (curved boxes),

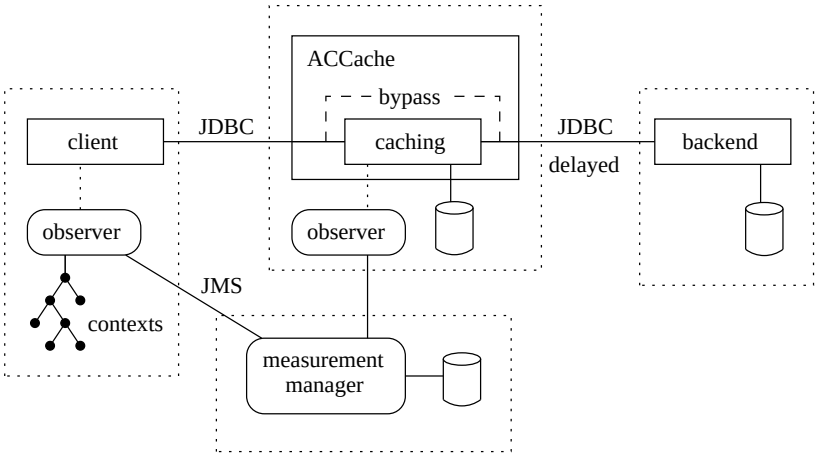

Fig. 1: Measurement setup on four nodes: client, ACCache, backend, and measurement manager

which are spread over four separate network nodes, is shown in Fig. 1. It also sketches two of our parameters that will be explained in the following: network delay and cache bypass.

3.3 Network Emulation

As it is difficult for us to actually maintain and use a backend DB in some remote part of the world, we employ a network emulator to approximate the characteristics of the network between backend and cache: NetEm is an enhancement of the traffic control facilities of the Linux kernel that allows adding delay, packet loss and other scenarios. [4]

The round-trip delay inherent in our real network between backend and cache node is about 0.2 ms. During our measurements we raised this round-trip delay by an amount of $\mu \pm \sigma$ according to a normal distribution with a standard deviation of $\sigma = \mu/10$ and a correlation $\rho = 25\%$. The mean round-trip delay μ was chosen from 0, 40, and 100 ms.

3.4 Bypassing Cache Functionality

As a baseline, we performed all measurements a second time with our cache still in the path from client to backend but with the main caching functionality turned off (i. e., no query analysis, probing, rewriting, etc., were performed but every query was immediately executed at the backend). In this case, our cache acted as a kind of forwarding proxy.

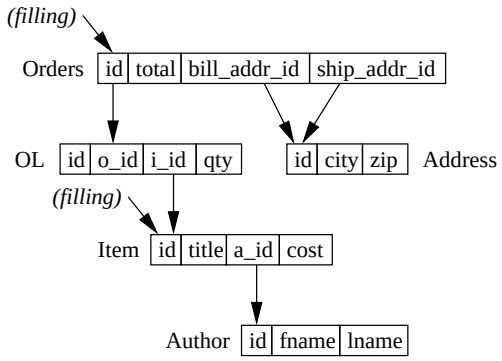

Fig. 2: A cache group for the TPC-W schema [5] (with five cache tables, two filling columns $O.id$ and $I.id$, and five RCCs)

3.5 Backend Schema and Cache Group

The scenario for our measurements is loosely based on the TPC-W benchmark [5], which models an online store. We use its database schema (with tables for customers, orders, items, etc.) and data in the backend DB (100000 items).

As a cache group, we use the one given in Fig. 2, which ensures that for any order loaded into the cache the corresponding order lines, addresses, and items are loaded, too. Furthermore, every item loaded into the cache will be accompanied by its author. Orders and items get into the cache only if referenced specifically by their primary keys (*id* columns).

3.6 Queries: Order Display

The queries that we pose to the cache are inspired by the web interaction "order display" of TPC-W. First of all, we display the details of a selected order including the referenced addresses:

```
select O.id, O.c_id, O.status, O.date, O.total, bill.*, ship.*
from orders O, address bill, address ship
where (O.bill_addr_id = bill.id) and (O.id = ⟨order id⟩)
  and (O.ship_addr_id = ship.id)
```

We then need a listing of all order lines belonging to that order where we include some basic information on the ordered items:

```
select OL.id, OL.qty, OL.discount, OL.comments, I.id, I.title, I.desc
from order_line OL, item I
where (OL.o_id = ⟨order id⟩) and (OL.i_id = I.id)
```

Finally, we simulate the user requesting the item details for each displayed order line in turn with multiple instances of the following statement.

```
select I.*, A.*
from item I, author A
where (I.id = ⟨item id⟩) and (I.a_id = A.id)
```

3.7 Measured Values

We designed two observers (for the client and cache), which transmit measurement values to the manager.

On the client, we have only a single execution context for executing a query. For each query, we capture three timestamps: before the query processing (create statement, execute statement, etc.) starts, when the first row of the query result has been fetched (first-row time), and after fetching and printing all resulting rows (all-rows time).

For the cache, we built the execution contexts "query", "analysis", and "load". In the query context, we capture the start and end timestamp of the query processing and a reference to the client query execution context that caused the execution on the ACCache system. One of the parts of processing a query is the analysis phase (probing, query rewriting). Therefore, an analysis context is created as a child context of the query and the start and the end of this phase are captured. Furthermore, the analysis phase might decide that tuples should be loaded into the cache: Each loading job created within our system is mirrored into a load execution context. This execution context captures the start and end timestamps and, additionally, the pair of column and value that is the starting point for the loading job.

Timestamps are retrieved using Java's `nanoTime` method, which has shown to have an accuracy of about ± 3 µs on our nodes. This means that the error in calculated durations will be twice that much.

4 Results

In our concrete setup, we executed the work unit "order display" five times in a row per measurement run without any delays between the queries: After displaying an order (O) and the retrieval of the corresponding order lines (OL), all of the related five items (I) were accessed. This work unit was then repeated for the very same order id.

As described above, we varied the round-trip delay between backend and cache and enabled or disabled our cache bypass: The six resulting configurations were repeated three times each, resulting in 18 measurement runs in total.

Figure 3 shows the average times spent on reading and displaying the query results in all measurement runs. Error bars indicate the spread of measured values (maximum and minimum).

Figure 4 shows the timing and the duration of client queries and load operations at the cache in a selected measurement run with a round-trip delay of 40 ms where the cache is not bypassed. The crosses mark significant points of time within the processing of a query, namely start of the query, the first-row

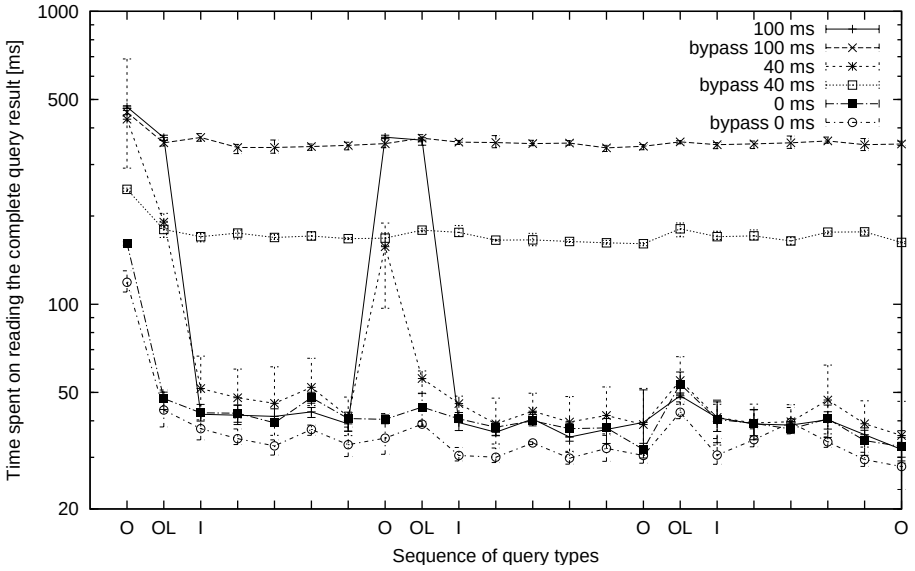

Fig. 3: Query execution times as perceived by the client

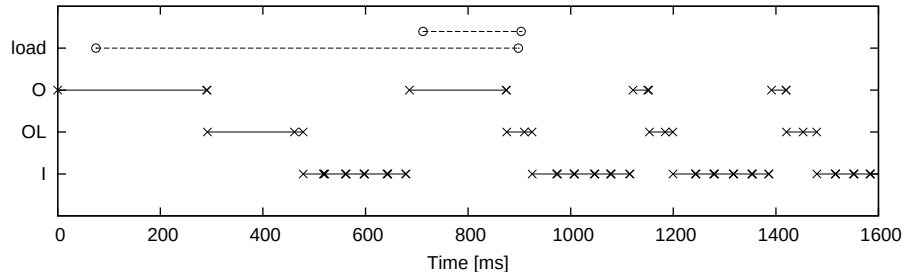

Fig. 4: Query execution/loading times and sequence of events (round-trip delay 40 ms, no bypass)

time, and the all-rows time. However, these are only visible separately in the case of an order-line query (OL). The other queries deliver only one row, which makes first-row time and all-rows time almost coincide.

As expected, the cache dramatically improves the response time of the queries if the cache loading for the order under consideration has finished. Interestingly, the cache can already be used to process the first five item queries when the loading has not yet finished (compare Figures 3 and 4). This is due to the fact that, in the current implementation, loading is performed bottom-up (as sketched in Sect. 2.3). That is, with our cache group, loading starts at the author and address tables and proceeds to the orders table. Therefore, the items related to the order requested become available (and usable) in the cache before the order itself does. As can be seen in Fig. 4, the loading is complete shortly after the second O query (which corresponds to the second set of peaks in Fig. 3); from then on, all following queries benefit from the cache contents.

Without delay (i. e., with a non-remote backend database), the cache needs 5 to 10 ms more time to answer a query than in the bypassing scenario. This is caused by the analysis phase and the probing in particular, which is always performed if a query could potentially be executed in the cache. When the delay rises, the costs involved in the probing are more than compensated by the savings due to the avoidance of remote accesses to the backend.

You might wonder why there are two loading operations in Fig. 4, which actually refer to the same order. The second loading operation is initiated at a time (ca. 700 ms) when the first operation has not yet succeeded in loading the order into the cache. When this first operation finishes at about 900 ms, a quick check suffices to see that there is not any work left. (Loading operations are executed strictly sequentially at the moment.)

5 Conclusion

We have subjected our ACCache prototype to a first series of simple measurements to get an indication of its potentials. Our results are encouraging: Already with small delays to the backend database server, our constraint-based cache is able to save query processing time, even for queries that are only related to an initiating query: Expected locality in database accesses can be conveniently modeled through cache groups, especially with cache constraints like RCCs that define an environment of related tuples.

We also learned from these measurements that setting up a general, automated measurement environment for a distributed system is a complex and time-consuming task. But after all, the possibility of designing well-suited execution contexts for tracing the work performed on the working nodes and their dependencies will assist us in setting up future measurement runs more quickly.

5.1 Outlook

There is a lot to be desired and to be done before we will have a full-featured transparent database cache that could prove its efficiency in measurements. An important aspect is updates.

At the moment, there is no support for updates in any part of our prototype: Updates of backend data are not propagated to the cache to update or invalidate stale cache data (which would have to respect the cache constraints, of course). Clients of our cache are unable to issue update statements to the database; it is unclear how these updates should be handled: sent to the backend, then propagated asynchronously back to the cache (the client might not see its own updates for a while); or directly applied to the cache and at the same time or later to the backend.

This brings up synchronization and consistency questions: How is the problem of accesses to multiple copies of backend data handled? Will there be locks, optimistic concurrency, or even some kind of context-specific conflict resolution with merging of changes? What level of transactional guarantees can be provided? May the client see out-of-date versions of the data? Should the application have to specify currency and consistency constraints acceptable to it [6]?

References

1. Härder, T., Bühmann, A.: Value complete, column complete, predicate complete – Magic words driving the design of cache groups. VLDB Journal (2007) Online First, http://dx.doi.org/10.1007/s00778-006-0035-9.
2. Bühmann, A., Härder, T., Merker, C.: A middleware-based approach to database caching. In Manolopoulos, Y., Pokorný, J., Sellis, T., eds.: ADBIS 2006. Volume 4152 of LNCS., Thessaloniki (2006) 182–199
3. Klein, J.: Development of an automated measurement environment for the constraint-based database caching (in German). Master's thesis, TU Kaiserslautern (October 2006) http://wwwdvs.informatik.uni-kl.de/pubs/DAsPAs/Kle06.DA.html.
4. Hemminger, S.: Network emulation with NetEm. In: Proceedings of linux.conf.au (LCA), Canberra (2005)
5. TPC: TPC benchmark W (web commerce) specification. http://www.tpc.org/tpcw/spec/tpcw_V1.8.pdf (2002) Version 1.8.
6. Guo, H., Larson, P.Å., Ramakrishnan, R., Goldstein, J.: Relaxed currency and consistency: How to say "good enough" in SQL. In Weikum, G., König, A.C., Deßloch, S., eds.: SIGMOD Conference, ACM (2004) 815–826