Caching with “Good Enough” Currency, Consistency, and Completeness

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

Per-Åke Larson
Microsoft
palarson@microsoft.com

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

ABSTRACT

SQL extensions that allow queries to explicitly specify data quality requirements in terms of currency and consistency were proposed in an earlier paper. This paper develops a data quality-aware, finer grained cache model and studies cache design in terms of four fundamental properties: *presence*, *consistency*, *completeness* and *currency*. The model provides an abstract view of the cache to the query processing layer, and opens the door for adaptive cache management. We describe an implementation approach that builds on the MTCache framework for partially materialized views. The optimizer checks most consistency constraints and generates a dynamic plan that includes currency checks and inexpensive checks for dynamic consistency constraints that cannot be validated during optimization. Our solution not only supports transparent caching but also provides fine grained data currency and consistency guarantees.

1. INTRODUCTION

Replicated data, in various forms, is widely used to improve scalability, availability and performance. Applications that use out-of-date replicas are clearly willing to accept results that are not current, but typically have some limits on how stale the data can be. SQL extensions that allow queries to explicitly specify such data quality requirements in the form of consistency and currency (C&C) constraints were proposed in [GLRG04]. That work also described how support for C&C constraints is implemented using MTCache [LGGZ04], a prototype mid-tier database cache built on Microsoft SQL Server.

We model cached data as materialized views over a primary copy. The work reported in [GLRG04] considered only the restricted case where all rows of a cached view are consistent, i.e., from the same database snapshot. This requirement severely restricts the cache maintenance policies that can be used. A *pull policy*, where the cache explicitly refreshes data by issuing queries to the source database, offers the option of using query results as the units for maintaining consistency and other cache properties. In particular, issuing the same parameterized query with different parameter values returns different partitions of a cached view, offering a much more flexible unit of cache maintenance (view partitions) than using entire views.

The extension to finer granularity cache management fundamentally changes every aspect of the problem, imposing non-trivial challenges: 1) how the cache tracks data quality; 2) how users specify cache properties; 3) how to maintain the cache efficiently; and 4) how to do query processing. In this paper, we propose a comprehensive solution described in Section 1.2.

Fig 1.1 shows our running example, where Q1 is a parameterized query, followed by different parameter settings.

1.1 Background and Motivation

We now motivate four properties of cached data that determine whether it can be used to answer a query. In the model proposed in [GLRG04], a query’s C&C constraints are stated in a currency clause. For example, in Q2, the currency clause specifies three “quality” constraints on the query results: 1) “ON (A, B)” means that all Authors and Books rows returned must be ***consistent***, i.e., from the same database snapshot. 2) “BOUND 10 min” means that these rows must be ***current*** to within 10 minutes, that is, at most 10 minutes out of date. 3) “BY authorId” means that all result rows with the same authorId value must be consistent. To answer the query from cached data, the cache must guaran-

```
Authors (authorId, name, gender, city, state)
Books (isbn, authorId, publisherId, title, type)

Q1:  SELECT * FROM Authors A WHERE authorId in (1,2,3)
     CURRENCY BOUND 10 min on (A) BY $key
E1.1: $key = Ø
E1.2: $key = authorId
E1.3: $key = city

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

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

Figure 1.1: Running example

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005

tee that the result satisfies these requirements and two more: 4) the Authors and Books rows for authors 1, 2, and 3 must be ***present*** in the cache and 5) they must be ***complete***, that is, no rows are missing.

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

Query Q3 illustrates the completeness property. It asks for all authors from Madison, but the rows for different authors do not have to be mutually consistent. Suppose we keep track of which authors are in the cache by their authorIds. Even if we have all the authors from Madison, we cannot use the cached data unless the cache guarantees that it has all the authors from Madison. Intuitively, the cache guarantees that its content is ***complete*** w.r.t. the set of objects in the master database that satisfy a given predicate.

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

1.2 Our Contributions

We offer a comprehensive solution to finer granularity cache management while still providing query results that satisfy the query's consistency and currency requirements. 1) We build a solid foundation for cache description by formally defining presence, consistency, completeness and currency *(Section 2)*. 2) We introduce a novel cache model that supports a specific way of partitioning and translate a rich class of integrity constraints (expressed in extended SQL DDL syntax) into properties required to hold over different partitions *(Section 3)*. 3) We identify an important property of cached views, called *safety*, and show how safety aids in efficient cache maintenance *(Section 4)*. Further, we formally define cache schemas and characterize when they are safe, offering guidelines for cache schema design *(Section 5)*. 4) We show how to efficiently enforce finer granularity C&C constraints in query processing by extending the approach developed in [GLRG04] *(Section 6)*. 5) We report experimental results, providing insight into various performance trade-offs *(Section 7)*.

2. CACHE PROPERTIES

The previous work in [GLRG04] describes the semantics of C&C constraints, providing a correctness standard. In this section, we define the properties of the cache using the same model. To be self-contained, we summarize the model and list some assumptions specific to this paper in Section 2.1.

2.1 Basic Concepts

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

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

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

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

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

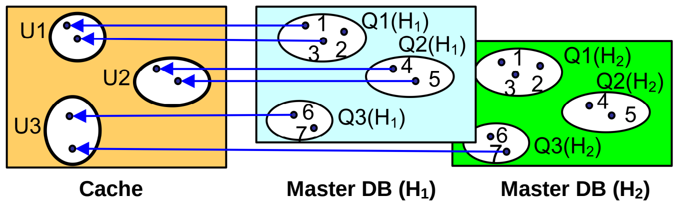

Figure 2.1: Cache property example

2.2 Presence

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

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

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

2.3 Consistency

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

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

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

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

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

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

Lemma 2.1: If a subset U of the cache S_{cache} is consistent w.r.t. a query Q and a snapshot H_n, then subset P(U) defined by any selection query P is consistent w.r.t. P°Q and H_n. □

Proof: See [GLR05] for all proofs omitted in this paper. □

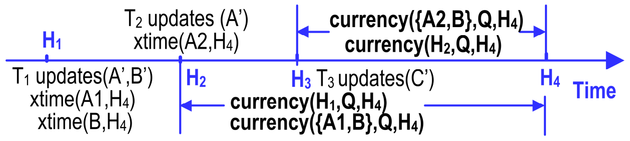

Figure 2.2: Currency example

2.4 Completeness

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

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

Lemma 2.2: If a subset U of the cache S_{cache} is complete w.r.t. a query Q and a snapshot H_n, then subset P(U) defined by any selection query P is complete w.r.t. P°Q and H_n. □

The above constraint is rather restrictive. Assuming that objects' keys are not modified, it is possible to allow subsequent updates of some objects in U to be reflected in the cache, while still allowing certain queries (which require completeness, but do not care about the modifications and can therefore ignore consistency) to use cached objects in U. See [GLR05] for *key-completeness* constraint.

Fig 2.1 illustrates cache properties, where an edge from object O to C denotes that C is copied from O. Assuming all objects are modified in H_2, U1 is consistent but not complete w.r.t. Q1 and H_1, U2 is complete w.r.t. Q2 and H_1, and U3 is key-complete w.r.t. Q3 and both H_1 and H_2.

Lemma 2.3: If a subset U of the cache S_{cache} is complete w.r.t. a query Q and a database snapshot H_n, then U is both key-complete and consistent w.r.t. Q and H_n. □

2.5 Currency

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

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

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

From the definition, the currency of U depends on the snapshot H_m used in the calculation. This can be solved using a "ghost row" technique, see [GLR05] for details.

Fig 2.2 illustrates the currency of two complete sets, where A1 and A2 are two copies of A' and B is a copy of B', $Q(H_i) = \{A', B'\}$, i = 1, 2, $Q(H_i) = \{A', B', C'\}$, i = 3, 4. {A1, B} and {A2, B} are complete w.r.t. Q and H_1, H_2.

3. DYNAMIC CACHING MODEL

In our model, a cache is a collection of materialized views $\mathbf{V} = \{V_1, \ldots, V_m\}$, where each view V_i is defined using a query expression Q_i. We describe the properties of the cache in terms of integrity constraints defined over **V**. In this section, we introduce a class of metadata tables called *control tables* that facilitate specification of cache integrity constraints, and introduce extended SQL DDL syntax for constraint specification. Fig 3.1 shows the set of DDL examples used in this section. We start by defining two views as shown in D1.

3.1 View Partitions and Control tables

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

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

```
D1: CREATE VIEW AuthorCopy AS      SELECT * FROM Authors
    CREATE VIEW BookCopy AS        SELECT * FROM Books
D2: CREATE TABLE AuthorList_PCT(authorId int)
    ALTER VIEW AuthorCopy ADD PRESENCE ON authorId IN
              (SELECT authorId FROM AuthorList_PCT)
D3: CREATE TABLE CityList_CsCT(city string)
    ALTER VIEW AuthorCopy ADD CONSISTENCY ON city IN
              (SELECT city FROM CityList_CsCT)
D4: CREATE TABLE CityList_CpCT(city string)
    ALTER VIEW AuthorCopy ADD COMPLETE ON city IN
              (SELECT city FROM CityList_CpCT)
D5: ALTER VIEW BookCopy ADD PRESENCE ON authorId IN
              (select authorId from AuthorCopy)
D6: ALTER VIEW BookCopy ADD CONSISTENCY ROOT
```

Figure 3.1: DDL examples for adding cache constraints

3.1.1 *Presence Control table (PCT)*

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

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

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

If V_i has at least one presence control table, it is a **partially materialized view (PMV)**, otherwise it is a fully materialized view addressed in [GLRG04]. See [ZLG05] for more general types of partial views, partial view matching, and run-time presence checking.

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

3.1.2 *Consistency Control table (CsCT)*

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

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

In our example, it is desirable to guarantee consistency for all authors from the same city, at least for some of the popular cities. We propose an additional `CONSISTENCY` constraint, for specifying this requirement. We first create a consistency control table containing a set of cities and then add a `CONSISTENCY` constraint to AuthorCopy, as in D3 of Fig 3.1. The `CONSISTENCY` clause specifies that the cache must keep all rows related to the same city consistent if the city is among the ones listed in CityList_CsCT; this is in addition to the consistency requirements implicit in the Presence Assumption. AuthorCopy can now be used to answer queries like E1.3.

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

3.1.3 Completeness Control table (CpCT)

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

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

A **completeness control table (CpCT)** for view V_i is denoted by CpCT[K]. A completeness control table is a consistency control table with an additional constraint: the subset U in V_i defined as before is not only consistent but also complete w.r.t. $(\sigma_{K=s.K} \circ Q_i)$ and H_n, for some snapshot H_n of the master database. We say K is a **completeness control-key** (**CpCK**). Note that all rows within the same completeness region must also be consistent (Lemma 2.3).

We propose to instruct the cache about completeness requirements using a `COMPLETENESS` constraint. Continuing our example, we create a completeness control table and then add a completeness clause to the AuthorCopy definition, as in D4 of Fig 3.1. Table CityList_CpCT serves as the completeness control table for AuthorCopy. If a city is contained in CityList_CpCT, then we know that either all authors from that city are contained in AuthorCopy or none of them are. Note that an entry in the completeness control table does not imply presence. Full completeness is indicated by dropping the clause starting with "IN". Not specifying a completeness clause indicates that the default completeness implicit in the Presence Assumption is sufficient.

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

3.2 Correlated Presence Constraints

In our running example, we may not only receive queries looking for some authors, but also follow-up queries looking for related books. That is, the access pattern to BookCopy is decided by the access pattern to AuthorCopy. In order to capture this, we allow a view to use another view as a presence control table. To have BookCopy be controlled by AuthorCopy, we only need to declare AuthorCopy as a presence control table by a `PRESENCE` constraint in the definition of BookCopy, as in D5 of Fig 3.1.

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

Circular dependencies require special care in order to avoid "unexpected loading", a problem addressed in [ABK+03]. In our model, we don't allow circular dependencies, as stated in Rule 1 in Fig 5.1. Thus we call a cache graph a **cache DAG**.

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

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

3.3 Correlated Consistency Constraints

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

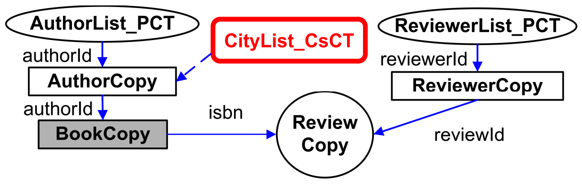

Figure 3.2: Cache schema example

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

If we wish the dependent view to be consistent with the controlling view, we add the consistency clause: `CONSISTENCY ROOT`, as in D6 of Fig 3.1. A node with such constraint is colored *gray*; it cannot have its own consistency or completeness control tables (Rule 2 in Fig 5.1).

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

Fig 3.2 illustrates a cache schema example, which consists of four partially materialized views. AuthorCopy is controlled by a presence control table AuthorList_PCT, likewise for ReviewerCopy and ReviewerList_PCT. Besides a presence control table, AuthorCopy has a consistency control table CityList_CsCT on city. BookCopy is both presence-wise and consistency-wise correlated to AuthorCopy. In contrast, ReviewCopy has two presence control tables: BookCopy and ReviewerCopy; it is view level consistent and consistency-wise independent from its parents.

4. SAFE CACHED VIEWS

A cache has to perform two tasks: 1) populate the cache and 2) reflect updates to the contents of the cache, while maintaining the specified cache constraints. Complex cache constraints can lead to unexpected additional fetches in a pull-based maintenance strategy, causing severe performance problems. We illustrate the problems through a series of examples, and quantify the refresh cost for unrestricted cache schemas in Theorem 4.1. We then identify an important property of a cached view, *safety*, that allows us to optimize pull-based maintenance, and summarize the gains it achieves in Theorem 4.2. We introduce the concept of *ad-hoc* cache regions, used for adaptively refreshing the cache.

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

4.1 Pull-Based Maintenance

In the "pull" model, we obtain a consistent set of rows using either a single query to the backend or multiple queries wrapped in a transaction. As an example, suppose AuthorCopy, introduced in Section 3, does not have any children in

Presence query:	**Consistency query:**	**Completeness query:**
SELECT * FROM Authors WHERE authorId = 1	SELECT * FROM Authors WHERE authorId in **K**	SELECT * FROM Authors WHERE city = "Madison"

Figure 4.1: Refresh query examples

Presence (Completeness) query:	**Consistency query:**
SELECT * FROM V WHERE K = k	SELECT * FROM V WHERE K_i in $\mathbf{K_i}$

Figure 4.2: Refresh queries

the cache DAG and that the cache needs to refresh a row t (1, Rose, Female, Madison, WI).

First, consider the case where AuthorCopy does not have any consistency or completeness control table, and so consistency follows the presence table. Then all rows in the presence region identified by authorId 1 need to be refreshed together. This can be done by issuing the presence query shown in Fig 4.1 to the backend server.

Next, suppose we have CityList_CsCT (see Section 3.1.2). If Madison is not found in CityList_CsCT, the presence query described above is sufficient. Otherwise, we must also refresh all other authors from Madison. If **K** is the set of authors in AuthorCopy that are from Madison, the consistency query in Fig 4.1 is sent to the backend server.

Finally, suppose we have CityList_CpCT (see Section 3.1.3). If Madison is found in CityList_CpCT, then besides the consistency query, we must fetch all authors from Madison using the completeness query in Fig 4.1.

Formally, given a cache region U<V, K, k>, let the set of presence control tables of V be $P_1, \ldots, P_n$, with presence control-keys $K_1, \ldots, K_n$. For K_i, i = 1..n, let $\mathbf{K_i}=\Pi_{K_i}\sigma_{K=k}(V)$, the remote queries for U are: 1) the presence query, if U is a presence region; 2) the consistency queries (i = 1..n), if U is a consistency region; and 3) the consistency queries (i = 1..n) (and the completeness query if $U \neq \emptyset$), if U is a completeness region. (The queries are shown in Fig 4.2.)

Lemma 4.1: For any cache region U <V, K, k> in the cache, the results retrieved from the backend server using the refresh queries in Fig 4.2 not only keeps U's cache constraints, but also keeps the presence constraints for the presence regions in V that U overlaps. □

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

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

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

For convenience, we assume that the calculation of AC(U) always eliminates consistency region U_i, if there exists a completeness region U_j in AC(U), s.t. $U_i = U_j$, since the completeness constraint is stricter (Lemma 2.3). The set of regions in AC(U) is partially ordered by the set containment relationship. From Lemma 2.1-2.3, we only need to maintain the constraints of some "maximal" subset of AC(U). Let **Max**(Ω) denote the set of the maximal elements in the partially ordered set Ω.

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

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

Maintenance Rule:

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

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

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

Theorem 4.1: Assuming the partial order between any two cache regions is constant, then given any region U, if we apply the Maintenance Rule to a cache instance that satisfies all cache constraints, let newTupleSet be the newly retrieved tuple set, Δ = AC(newTupleSet), then

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

The last part of the theorem shows that when a region U is refreshed, every region in MaxAC(U) must be simultaneously refreshed. Otherwise, there is some instance of the cache that satisfies all constraints, yet running the refresh transaction to refresh U will leave the cache in a state violating some constraint. If (Δ-Ω)≠Ø, multi-trip to the master database is needed in order to maintain all cache constraints.

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

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

The potentially expensive calculation and the large size of MaxAC(U), and hence the high cost of refreshing the cache motivate the definition of *safe* views in Section 4.2.

4.1.1 Ad-hoc Cache Regions

Although the specified cache constraints are the minimum constraints that the cache must guarantee, sometimes it is desirable for the cache to provide additional "ad-hoc" guarantees. For example, a query workload like E1.1 asks for authors from a set of popular authors and requires them to be mutually consistent. Popularity changes over time. In order to adapt to such workloads, we want the flexibility of grouping and regrouping authors into cache regions on the fly. For this purpose, we allow the cache to group regions into *"ad-hoc" cache regions*. See [GLR05] for details.

4.1.2 Keeping Track of Currency

When using the pull model, we keep the last refresh timestamp for each cache region. If current time is *t*, a region with timestamp *T* is no older than (*t – T)*, since all updates until *T* are reflected in the result of the refresh query.

4.2 Safe Views and Efficient Pulling

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

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

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

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

Lemma 4.2: For a safe white PMV V that doesn't have any children, given any cache region U in V, the partially ordered set AC(U) is a tree. □

Since AC(U) on V has a regular structure, we can maintain metadata to find the maximal element efficiently. We omit the detailed mechanism because of space constraints.

Theorem 4.2: Consider a white PMV V, and let κ denote V and all its gray descendants. If all nodes in κ are safe, whenever any region U defined on V is to be refreshed:

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

5. DESIGN ISSUES FOR CACHES

In this section, we investigate conditions that lead to unsafe cached views and propose appropriate restrictions on allowable cache constraints. In particular, we develop three additional rules to guide cache schema design, and show that

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

Figure 5.1: Cache schema design rules

Rules 1-5 are a necessary and sufficient condition for (all views in) the cache to be safe.

5.1 Shared-Row Problem

Let's take a closer look at the AuthorCopy and BookCopy example defined in Section 3. Suppose a book can have multiple authors. If BookCopy is a rectangle, since co-authoring is allowed, a book in BookCopy may correspond to more than one control-key (authorId) value, and thus belong to more than one cache region. To reason about such situations, we introduce cache-instance DAGs.

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

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

There are two cases where a view V has the shared-row problem. In the first case (Lemma 5.1), we can only eliminate the potential overlap of regions in V defined by different presence control tables if V is view-level consistent. Considering the second condition in the definition of safe, we have Rule 3 in Fig 5.1. For the second case (Lemma 5.2) we enforce Rule 4 in Fig 5.1.

Lemma 5.1: Given a cache schema <**W**, **E**>, view V in **W** has the shared-row problem if V has more than one parent.□

Lemma 5.2: Given a cache schema <**W**, **E**>, for any view V, let the parent of V be V_1. V has the shared-row problem iff the presence key K in V_1 for V is not a key in V_1. □

5.2 Control table Hierarchy

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

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

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

Rule 5 is stated in Fig 5.1. We require that a new cache constraint can only be created in the system if its addition does not violate Rules 1-5.

Theorem 5.1: Given a cache schema **<W**, **E>**, if it satisfies rules 1-5, then every view in **W** is safe. Conversely, if the schema violates one of these rules, there is an instance of the cache satisfying all specified integrity constraints in which some view is unsafe. □

6. ENFORCING C&C CONSTRAINTS

A traditional distributed query optimizer decides whether to use local data based on data availability and estimated cost. In our setting, it must also take into account local data properties (presence, consistency, completeness and currency). Presence checking is addressed in [ZLG05]; the same approach can be extended to completeness checking. This section describes efficient checking for C&C constraints in a transformation-based optimizer. See [GLR05] for proofs.

In comparison to [GLRG04], the algorithms developed here are more general and support finer granularity C&C checking. In [GLRG04], consistency checking was done completely at optimization time and currency checking at run time, because view level cache region information is stable and available at optimization, while currency information is only available at run time. In this paper we still perform as much as possible of the consistency checking at optimization time but part of it may have to be delayed to run-time. For a view with partial consistency guarantees, we don't know at optimization time which actual groups will be consistent at run time. Further, ad-hoc cache regions may change over time, also prompting run-time checking.

6.1 Normalizing C&C Constraints

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

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

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

All constraints from individual currency clauses are merged together into a single constraint and converted into an equivalent or stricter normalized form with no redundant requirements. See [GLR05] for details.

6.2 Compile-time Consistency Checking

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

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

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

6.2.1 Required Consistency Plan Property

A query's required consistency property consists of the normalized consistency constraint described in section 6.1.

6.2.2 Delivered Consistency Plan Property

A delivered consistency property CPd consists of a set of tuples $\{<R_i, \mathbf{S}_i, \mathbf{\Omega}_i>\}$ where R_i is the id of a cache region, $\mathbf{S}_i$ is a set of input operands, namely, the input operands of the current expression that belong to region R_i, and $\mathbf{\Omega}_i$ is the set of grouping keys for the input operands. Each operator computes its delivered plan properties bottom-up based on the delivered plan properties of its inputs. We omit the algorithms due to space constraints; for details see [GLR05].

6.2.3 Satisfaction Rules

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

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

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

Secondly, a complete plan satisfies the constraint if each required consistency group is fully contained in some delivered cache region. We extend the consistency satisfaction rule in [GLRG04] to include finer granularity cache regions.

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

For query Q2, suppose we have CCr = {<5, Ø, {Authors, Books}, {isbn}>}, and that the cache schema is the one in Fig 3.2. During view matching, AuthorCopy and BookCopy will match Q2. Thus CPd = {<-1, {Authors, Books}, {Authors.authorId, city}>}. If AuthorCopy joins with BookCopy on authorId (as indicated by the presence correlation), and the result is R, then from the key constraints of Authors and Books we know that isbn is a key in R. Therefore city$\in \{\text{isbn}\}^+$. CPd satisfies CCr.

While a plan is being constructed, bottom-up, we want to stop as soon as it is possible when the current subplan cannot deliver the consistency required by the query. The consistency satisfaction rule cannot be used for checking subplans; a check may fail simply because the partial plan does not include all inputs covered by the required consistency property. Instead we apply the following *violation rules*. We prove that a plan cannot satisfy the required plan properties if a subplan violates any of the three rules [GLR05].

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

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

6.3 Run-time C&C Checking

To include C&C checking at runtime, the optimizer must produce plans that check whether a local view satisfies the

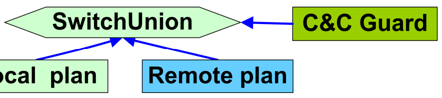

Figure 6.1: SwitchUnion with a C&C guard

required C&C constraints and switch between using the local view and retrieving the data from the backend server. Such run-time decision-making is built in a plan by using a *SwitchUnion* operator. A SwitchUnion operator has multiple input streams but only one is selected at run-time based on the result of a selector expression.

In MTCache, all local data is defined as materialized views and logical plans making use of a local view are always created through view matching [LGZ04, GL01]. Consider an (logical) expression E and a matching view V from which E can be computed. If C&C checking is required, we produce a substitute consisting of a SwitchUnion on top, shown in Fig 6.1, with a selector expression that checks whether V satisfies the currency and consistency constraint and two input expressions: a local branch and a remote branch. The local branch is a normal substitute expression produced by view matching and the remote plan consists of a remote SQL query created from the original expression E. If the condition, which we call consistency guard or currency guard according to its purpose, evaluates to true, the local branch is chosen, otherwise the remote one.

The discussion of when and what type of consistency checking to generate and the inexpensive consistency checking we support is deferred to Section 7.

7. PERFORMANCE STUDY

This section reports experimental results for consistency checking; results for presence and currency checking are reported in [ZLG05] and [GLRG04] respectively.

7.1 Experimental Setup

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

```
Settings:  CREATE TABLE C_PCT (ckey int PRIMARY, rid int)
           CREATE TABLE C_CsCT(nkey int PRIMARY, rid int)
           CREATE TABLE O_PCT (ckey int PRIMARY, rid int)
Qa:  SELECT *  FROM  customer C
     WHERE c_custkey in $custSet
     [CURRENCY BOUND 10 on (C) BY $key]
Qb:  SELECT * FROM customer C, orders O
     WHERE c_custkey=o_custkey and c_custkey in $custSet
     [CURRENCY BOUND 10 on (C, O) BY $key]
Qc:  SELECT * FROM customer C
     WHERE c_nationkey in $nationSet
     [CURRENCY 10 on (C) BY $key]
```

Figure 7.1: Settings & Queries used for experiments

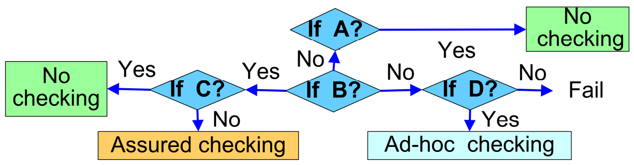

Figure 7.2: Generating consistency guard

used are shown in Fig 7.1. We populated the ckey and nkey columns with c_custkey and c_nationkey columns from the views respectively.

C_PCT and O_PCT are the presence control tables of CustCopy and OrderCopy respectively. C_CsCT is a consistency control table on CustCopy. By setting the timestamp field, we can control the outcome of the consistency guard.

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

7.2 Consistency Guard Overhead

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

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

```
A11a, A11b:  SELECT 1 WHERE NOT EXISTS (
        SELECT 1 FROM CustCopy
        WHERE c_custkey IN $custSet
        GROUP BY c_nationkey
        HAVING [COUNT(*)>1 AND] c_nationkey NOT IN
            (SELECT nkey FROM C_CsCT) )
A12: SELECT 1 WHERE |$nationSet|      =  (
         SELECT COUNT(*) FROM C_CsCT
         WHERE  nkey IN $nationSet)
S11: SELECT 1 WHERE 1 = (
         SELECT COUNT(DISTINCT rid) FROM C_PCT
         WHERE  ckey IN $custSet )
S12: SELECT 1 WHERE 1 =
         ALL (SELECT COUNT(DISTINCT rid) FROM C_PCT, CustCopy
                 WHERE c_custkey IN $custSet AND ckey=c_custkey
                 GROUP BY c_nationkey)
S21: SELECT 1 FROM(
         SELECT COUNT (DISTINCT rid1) AS count1,
                 SUM (ABS(rid1-rid2)) AS count2
         FROM  ( SELECT   A.rid AS rid1, B.rid AS rid2)
                 FROM     C_PCT A, O_PCT B
                 WHERE    A.ckey IN $custSet AND
                          A.ckey = B.ckey) ) AS FinalCount
         WHERE count1 = 1 AND count2 = 0)
S22: SELECT 1 WHERE NOT EXISTS (SELECT 1 FROM
         ( SELECT     c_custkey,c_nationkey,
                      A.rid AS rid1, B.rid AS rid2
           FROM       C_PCT A, O_PCT B, CustCopy C
           WHERE      A.ckey IN $custSet AND
                      A.ckey = c_custkey AND c_custkey = B.ckey
         ) AS FinalCount
         GROUP BY c_nationkey
         HAVING (MIN(rid1) <> MAX(rid1) OR
            MIN(rid2) <> MAX(rid2) OR MIN(rid1) <> MIN(rid2)))
```

Figure 7.3: A spectrum of consistency guards

Cost	Local			Remote		
	Qa	Qb	Qc	Qa	Qb	Qc
ms	.078	.08	1.17	.01	.19	1.13
%	16.56	14.00	<2	<1	<2	<1
# Rows	1	6	5975	1	6	5975

Table 7.1: Simple consistency guard overhead

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

Table 7.2: Single-table case overhead

7.2.1 Single Table Case

We first analyze what type of consistency guard is needed for Qa when $key differs. The decision making process is shown in Fig 7.2 and the consistency guards in Fig 7.3.

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

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

Condition B: Is each required consistency group equal to or contained by a consistency region?

If Yes, we check C, otherwise we check D.

Condition C: Is the consistency guarantee full?

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

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

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

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

If Yes, we have the opportunity to do ad-hoc consistency checking. For Qa with $key = Ø, we check if all the required customers are in the same ad-hoc cache region (S11). Such checking (e.g., S11, S12 and S21, S22 from Section 7.1.2) is called **ad-hoc consistency checking**.

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

Experiment 1 is designed to measure the overhead of the simple consistency guards supported in our current framework. We choose to support only run-time consistency guards that 1) do not require touching the data in a view; 2)

Cost	Local		Remote	
	S21	S22	S21	S22
ms	.90	.83	1.00	.98
%	155.83	143.82	24.82	24.36

Table 7.3: Multi-table case overhead

only require probing a single control table. We fixed the guards and measured the overhead for: Qa and Qb with $custSet = (1); Qc with $nationSet = (1). The consistency guard for Qa and Qb is S11 and the one for Qc is A12.

The results are shown in Table 7.1. As expected, in both the local and remote case, the absolute cost remains roughly the same, the relative cost decreases as the query execution time increases. The overhead for remote execution is small (< 2%). In the local case, the overhead for Qc (returning ~6000 rows) is less than 2%. Although the absolute overhead for Qa and Qb is small (<0.1ms), since the queries are inexpensive (returning 1 and 6 rows respectively), the relative overhead is ~15%.

In experiment 2, we used query Qa with $custSet = (2, 12), which returns 2 rows; and compared the overhead of different types of consistency guards that involve one control table. The results are shown in Table 7.2.

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

7.2.2 Multi-Table Case

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

Condition E : Do they have the same consistency root?

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

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

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

In Experiment 3, we use query Qb with $custSet = (2, 12), which returns 7 rows, and measure the overhead of consistency guards that involve multiple control tables. The results are shown in Table 7.3. Guards S21 and S22 involve not only accessing the data, but also performing joins. Such complicated checking incurs huge overhead in the local execution case (~150%). Note that if CustCopy and OrderCopy are consistency-wise correlated, then the overhead (refer to single-table case) reduces dramatically.

8. RELATED WORK

The work in [GLRG04] is the first that addresses C&C aware database caching with a query centric approach. Relaxing data quality is an old concept in replica management, distributed databases and warehousing and web views. Some authors take a maintenance-centric approach [ABG88, GN95, SK97], where queries are not allowed to express their individual data quality requirements. Others have taken a query-centric approach [OW00, HSW94, WXCJ98, OLW01], but they focus on single object granularity and no consistency guarantee is provided. FAS [RBSS02] enforces consistency at the level of the complete cache. In concurrency control, Epsilon-serializability [PL91] allows higher degree of concurrency by relaxing data quality.

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

The closest works to ours are DBCache [ABK+03] and Constraint-based Database Caching (CBDC) [HB04]. Similarly to us, they consider full-fledged DBMS caching; and they define a cache with a set of constraints. However, there are two fundamental differences. First, they don't consider relaxed data quality requirements, nor do they provide currency guarantees from the DBMS. Our work is more general in the sense that the cache-key and RCC constraints (an extension to cache groups in [TT02]) they support can be seen as a subset of ours. Second, in DBCache, local data availability checking is done outside of the optimizer, while in our case, local data checking is integrated into query optimization, which not only allows finer granularity checking, but also leaves the optimizer the freedom to choose the best plan based on cost.

9. CONCLUSIONS

The goal of our work is to build a solid foundation for fine granularity, C&C-aware adaptive DBMS caching. We formally defined four fundamental cache properties: presence, consistency, completeness, and currency. We proposed a cache model in which users can specify a cache schema by defining a set of local views, together with cache constraints that specify what properties the cache must guarantee. We enforced C&C constraints by integrating C&C checking into query optimization and evaluation.

We envision three lines of future research. First, in our current cache model, we only support groups defined by equality conditions. For efficient cache management, we plan to explore other predicates, e.g., range predicates. Second, we plan to investigate C&C-aware cache replacement and refresh policies that make decisions adaptively, based on the workload. Third, we want to conduct a holistic system performance study to evaluate the effectiveness of different design choices in relaxed C&C database caching. One example would be cache management granularity: view-level (as described in [GLRG04]) vs. group-level cache regions.

10. REFERENCES

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

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

[AJL+02] J. Anton, L. Jacobs, X. Liu, J. Parker, Z. Zeng, T. Zhong. Web Caching for Database Applications with Oracle Web Cache. *SIGMOD'02*.

[APTP03] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A Dynamic Data Cache for Web Applications. *ICDE*, 2003.

[CLL+01] K. S. Candan, W. Li, Q. Luo, W. Hsiung, and D. Agrawal. Enabling Dynamic Content Caching for Database-Driven Web Sites. *SIGMOD*, 2001.

[DDT+01] A. Datta, K. Dutta, H. Thomas, D. VanderMeer, K. Ramamritham, D. Fishman. A Comparative Study of Alternative Middle Tier Caching Solutions to Support Dynamic Web Content Acceleration. *VLDB*, 2001.

[DFJ+96] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava and M. Tan, Semantic Data Caching and Replacement. *VLDB*, 1996.

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

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

[GLR05] H. Guo, P. Larson, R. Ramakrishnan. Caching with "Good Enough" currency, consistency, and completeness". TR1520, University of Wisconsin, 2005. http://cs.wisc.edu/~guo/publications/TR1450.pdf

[GLRG04] H. Guo, P. Larson, R. Ramakrishnan, J. Goldstein: Relaxed Currency and Consistency: How to Say "Good Enough" in SQL. *SIGMOD*, 2004.

[HB04] T. Härder, A. Bühmann. Query Processing in Constraint-Based Database Caches. *Data Engineering Bulletin* 27(2), 2004.

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

[KB96] A. Keller, J. Basu. A Predicate-Based Caching Scheme for Client-Server Database Architectures. *VLDB J.* 5(1):35-57, 1996.

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

[LKM+02] Q. Luo, S. Krishnamurthy, C.Mohan, H. Woo, H. Pirahesh, B. G. Lindsay, J. F. Naughton. Middle-tier database caching for e-Business. *SIGMOD*, 2002.

[LN01] Q. Luo and Jeffrey F. Naughton. Form-Based Proxy Caching for Database-Backed Web Sites". *VLDB* 2001.

[Moh01] C. Mohan. Caching Technologies for Web Applications. *VLDB*, 2001.

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

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

[TT02] The TimesTen Team. Mid-tier Caching: The FrontTier Approach. *SIGMOD*, 2002.

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

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

[SK97] L. Seligman and L. Kerschberg. A Mediator For Approximate Consistency: Supporting "Good Enough" Materialized Views. *JIIS*, 1997.

[WXCJ98] O.Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues And Solutions. SSDBM, 1998.

[ZLG05] J. Zhou, P. Larson, J. Goldstein. Partially Materialized Views. MSR-TR-2005-77, Microsoft Research, 2005. ftp://ftp.research.microsoft.com/pub/tr/TR-2005-77.pdf