University of Wisconsin Technical Report (TR1520) 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*. Such a 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 plan compilation. Our solution not only supports transparent caching but also provides transactional 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 Applications that use out-of-date replicas are clearly willing to accept results that are not current, but typically have some limits on how stale the data can be.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, as described in Section 1.2.

Figure 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: i) "ON (A, B)" means that any Authors and Books rows joined together must be ***consistent***, i.e., from the same database snapshot. ii) "BOUND 10 min" means that these rows must be ***current*** to within 10 minutes, that is, at most 10 minutes out of date. iii) "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 guarantee that the result satisfies these requirements and two more: iv) the Authors and Books rows for authors 1, 2, and 3 must be ***present*** in the cache and v) 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 contents are ***complete*** w.r.t. the set of objects in the master database that satisfy a given predicate.

```
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

No matter what cache management mechanisms or policies are used, as long as cache properties are accurately maintained, query processing can deliver correct results. Thus, cache property descriptions serve as an abstraction layer between query processing and cache management, which enables the implementation of query processing to be independent of cache maintenance.

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 analytical and 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 make this paper more 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_i, i = 1..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 "$\circ$" 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 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 limit our discussion to selection queries, and we only consider views defined by selection queries that select a subset of data from a table or a view of the master database.

Self-Identification: The function **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 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 copy C in snapshot H_n is measured by how long it has been stale, i.e., **currency**(C, H_n) = xtime(T_n) - **stale**(C, H_n).

2.2 Presence

The simplest type of query asks for an object identified by its key, as shown in Q1. How do we know an object is in the cache?

Intuitively, we require that every object in the cache must be copied from some valid database snapshot. The function **return**(O, s) returns 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 database snapshot H_n 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**) We say an object O is present in cache S_{cache} iff there is a copy C in S_{cache} such that master(C) = O, and for some snapshot H_n of the master database 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 if and only if 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 E.1.3. 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 such that CopiedFrom(U, H_n) holds, and for some query Q, the following holds: $U \subseteq Q(H_n)$, then U is **snapshot consistent** (or **consistent**) w.r.t. Q and H_n. □

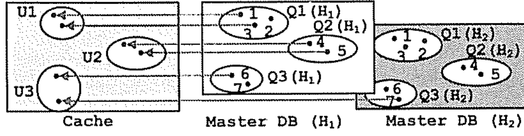

Figure 2.1: Cache property example

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: Given a subset of objects U in the cache S_{cache}, if U is consistent w.r.t. a query Q and a snapshot H_n of the master database, then any subset P(U) defined by a selection query P is consistent w.r.t. P°Q and H_n. □

Proof: since U is consistent w.r.t. Q and H_n, we have:

$U \subseteq Q(H_n)$ (1)

CopiedFrom(U, H_n). (2)

Since (1), for any selection query P,

$P(U) \subseteq P \circ Q(H_n)$ (3)

Since P is a selection query, $P(U) \subseteq U$. Together with (2), we have

CopiedFrom(P(U), H_n). (4)

From (3) and (4), we know that P(U) is snapshot consistent w.r.t. P°Q and H_n. □

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 if and only if CopiedFrom(U, H_n) holds and U = $Q(H_n)$. □

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

Proof: from the given, we have

CopiedFrom(U, H_n), (1)

U = $Q(H_n)$ (2)

From (2), for any selection query P,

P(U) = P°Q(H_n) (3)

Since $P(U) \subseteq U$, from (1), we have

CopiedFrom(P(U), H_n) (4)

From (3) and (4), we know P(U) is complete w.r.t. P°Q and H_n. □

The completeness constraint is rather restrictive. If we assume that objects' keys are not modified, then 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.

Defn: (**Associated Objects**) We say that a subset U of the cache S_{cache} is **associated with** a query Q if for each object C in U, there exists a snapshot H_n of the master database such that CopiedFrom(C, H_n) holds and C is in Q(H_n). □

Defn: (**Key-completeness**) For a subset U of the cache S_{cache}, we say U is key-complete w.r.t. Q and a snapshot H_n, iff U is associated with Q, and $\Pi_{key}Q(H_n) \subseteq \Pi_{key}(U)$. □

Intuitively, U includes (as identified by the keys) all the objects that appear in the result of Q applied to the master database H_n. However, the objects in the cache might have been copied from different earlier snapshots of the master database, and subsequent changes to these objects might not be reflected in the cache.

Figure 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. □

Proof: Directly from the definitions. □

2.5 Currency

We have defined *stale point* and *currency* for a single object. Now we extend the concepts to a subset of objects. Suppose that on day 1, there are only two authors from Madison in the master database, and we copy them to the cache, forming set U. On day 2, a new author moves to Madison. On day 3, how stale is U w.r.t. predicate "city = Madison"? Intuitively, the answer should be 1 day, 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 continue to be current. To solve this problem, we use the snapshot where U is copied from as a reference.

We overload the stale() function 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**) For a subset U of the cache S_{cache}, if U is complete w.r.t. a query Q and a database snapshot H_m, then the currency of U w.r.t. a snapshot H_n of the master database is defined as follows: currency(U, Q, H_n) = currency(H_m, Q, H_n). □

From the definition, it seems the currency of U depends on which snapshot H_m we use in the calculation. In order to avoid such ambiguity, we introduce the following assumption. The assumption can be relaxed by a "ghost row" technique, see [GLR05] for details.

<u>**Non-Shrinking Assumption:**</u> For any query Q, any database snapshot H_i and H_j, where i≤j, and $\Pi_{key}Q(H_i) \subseteq \Pi_{key}Q(H_j)$. □

Currency Property 2.1: Under the assumption above, for any subset U of the cache S_{cache}, any query Q, and any master database

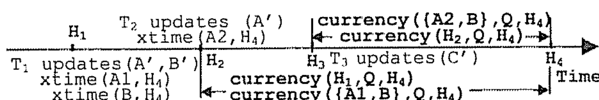

Figure 2.2: Currency example (1)

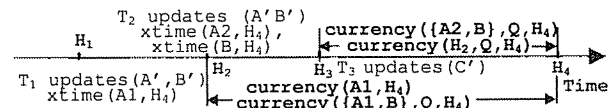

Figure 2.3: Currency example (2)

snapshot H_i and H_j, if U is complete w.r.t. Q and both H_i and H_j, then for any n, currency(H_i, Q, H_n) = currency(H_j, Q, H_n). □

Proof: (by contradiction) Since the case i=j is trivial, without loss of generality, assume i<j. Assume T_k is the first transaction that modifies Q(H_i) after H_i. We claim that k>j. For the proof by contradiction, assume k≤j.

From the non-shrinking assumption, T_k either 1) modifies an object in Q(H_i), say O1 or 2) adds a new object, say O2 to the result of Q. Further, both O1 and O2 are in Q(H_j).

In case 1), since k≤j, xtime(O1, H_j)>xtime(O1, H_i), which contradicts the given that U is consistent w.r.t. both H_i and H_j.

In case 2), O2 is not in Q(H_i), which also contradicts the given that U is complete w.r.t. both H_i and H_j.
Thus k>j, hence currency(H_i, Q, H_n) = currency(H_j, Q, H_n). □

Figure 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'} for i = 1, 2, Q(H_i) = {A', B', C'} for i = 3, 4. Thus {A1, B}and {A_2, B} are complete w.r.t. Q and H_1, H_2 respectively.

How to measure the currency of a key-complete set? Figure 2.3 shares the same assumptions as Figure 2.2, except for T_2 and xtime(B), where {A1, B}and {A_2, B} are key-complete w.r.t. Q and H_1 and H_2, while the latter is also complete w.r.t. Q and H_2. It is desirable that 1) currency({A1,B}, Q, H_4) is deterministic; and 2) Since A1 is older than A2, {A1, B}should be older than {A2, B}.

We address these problems by firstly identifying a unique referenced snapshot, and secondly incorporating the currency of the objects into the currency definition.

Defn: (**Max key-complete snapshot**) For any subset U of the cache S_{cache} and a query Q, the max key-complete snapshot of U w.r.t. Q and a database snapshot H_n, **max-snapshot**(U, Q, H_n) is equal to H_k, if there exists k, s.t., for any i≤k, $\prod_{key} Q(H_i) \subseteq \prod_{key} U$

And one of the following conditions holds: 1) k=n; 2) $\prod_{key} U \subset \prod_{key} Q(H_{k+1})$

Otherwise it is Ø. □

Directly from the definition of key-completeness and the non-shrinking assumption, we have the following lemma.

Lemma 2.4: If there exists a database snapshot H_m, s.t. U is key-complete w.r.t. Q and H_m, then for any n, max-snapshot(U, Q, H_n) is not Ø. □

Lemma 2.4 guarantees that the following definition is well defined for a key-complete set.

Defn: (**Currency for key-complete set**) For a subset U of the cache S_{cache}, if U is key-complete w.r.t. a query Q and some database snapshot, then the currency of U w.r.t. a snapshot H_n of the master database is defined as follows. Let H_m = max-snapshot(U, Q, H_n) and

$$Y = \max_{C \in U} (currency\ (C, H_n)),$$

Then Currency(U, Q, H_n) = max (Y, currency(H_m, Q, H_n)). □

Figure 2.3 shows the currency of a key-complete set {A1. B} and a complete set {A2, B}.

Now the currency of a key-complete set has some nice properties that fit in intuition.

Currency Property 2.2: For any subset U of the cache S_{cache}, and a query Q, if U is key-complete w.r.t. Q and some database snapshot, then for any n, currency(U, Q, H_n) is deterministic. □

Proof: Directly from the definition and Lemma 2.4. □

Currency Property 2.3: Given any query Q, and two subsets U1 and U2 of the cache S_{cache}, if max-snapshot(U1, Q, H_n) = max-snapshot(U2, Q, H_n) ≠ Ø, let

$$Y_i = \max_{O \in U_i} (currency\ (O, H_n)),$$

where i=1, 2. If $Y_1 \geq Y_2$, then currency(U1, Q, H_n)≥currency(U2, Q, H_n). □

Proof: directly from the definition. □

Currency Property 2.4: currency-complete is a special case of currency-key-complete. □

Proof: Given any subset U of the cache S_{cache} that is complete w.r.t. a query Q and some database snapshot H_m. For any n≥m, let H_g = max-snapshot(U, Q, H_n). From the definition of max key-complete snapshot we know g≥m. There are two cases:

Case 1: U is complete w.r.t. H_g.
Let T_k be the first transaction in H_n that changes the result of Q after H_g. From the non-shrinking assumption, again, we have two cases:

a. T_k touches at least one object, say O1, in U. Since T_k is the first transaction that touches U,

$$Y = \max_{O \in U} (currency\ (O, H_n)) = currency\ (O_1, H_n)$$

Since the stale points for O1 and Q(H_g) are both xtime(T_k), currency(H_g, Q, H_n) = currency(O1, H_n). Thus
currency(U, Q, H_n) = max (Y, currency(H_g, Q, H_n))
= currency(H_g, Q, H_n) = currency(O1, H_n).

b. T_k adds new objects into the result of Q.
In this case the stale point of any object O in U is later than xtime(T_k), so currency(H_g, Q, H_n) ≥ currency(O, H_n).
currency(U, Q, H_n) = max (Y, currency(H_g, Q, H_n))
= currency(H_g, Q, H_n).

Case 2: U is not complete w.r.t. H_g.
let T_k be the first transaction in H_n that modifies at least an object, say O1 in U after H_m, then

currency(H_m, Q, H_n) = currency(O1, H_n) (2)

$$Y = \max_{O \in U} (currency(O, H_n)) = currency(O_1, H_n) \quad (3)$$

In addition we have k≤g, otherwise from the non-shrinking assumption, U would be complete w.r.t. H_g. Thus

Y ≥ currency(H_g, Q, H_n) (4)

Putting (2), (3) and (4) together,
currency(U, Q, H_n) = max (Y, currency(H_g, Q, H_n))
= currency(H_g, Q, H_n) = currency(O1, H_n). □

2.6 Dealing with Deletion

Currency properties 2.1 to 2.4 don't hold without the non-shrinking assumption. Take Property 2.1 for example. On day 1 there are two customers C1, C2 from WI, which we copied to the cache, U = {C1, C2}. On day 2, customer C3 moved to WI temporarily, and moved out of WI on day 5. Then on day 4, the currency of U is 2 days old. However, on day 6, it goes back to 0!

The reason is that when an object is deleted, we lose its xtime record. Consequently, given a set of objects **K**, one cannot

uniquely identify the first snapshot **K** appears in. To remedy that, we introduce the concept of **ghost object**. Conceptually, when an object is deleted from a region in the master copy, we don't really delete it, instead, we mark it as a ghost object and treat it the same way as a normal object. Thus we keep the xtime timestamp of deleted objects. Ghost objects and their timestamps are propagated to the cache just as normal objects. With this technique, deletion is modeled as a special modification. Thus the non-shrinking assumption is guaranteed even in the presence of deletions.

Lemma 2.5: With the ghost object technique, given any query Q, the non-shrinking assumption holds. □

Proof: With the ghost object technique, there are no deletions to the region defined by Q. □

Note that in practice, we don't need to record those ghost objects, since the calculation of currency only needs to be conservative. How we bound the currency of a complete set is discussed in Section 4.1.2.

2.7 Derived Data

If the cache only contains (parts of) base tables, then for each object in the cache there is a master version in the master database. This doesn't apply to derived data, i.e., materialized views in the cache. An object (row) in a materialized view in the cache doesn't have a master copy in the master database. We introduce the concept of ***virtual master copy*** to remedy this. Conceptually, for any view V in the cache, for any snapshot H_i of the master database, we calculate $V(H_i)$ and include it in the master database. Thus by comparing two adjacent snapshots, we can record any insertion/deletion/modification on the view. With this technique, any object in the cache — no matter whether it is from a base table or a view — has a master copy in the master database. Thus, any query can be used to define a region in the cache.
Again, in practice, since we only need to bound the currency of a region conservatively, we don't need to materialize the virtual master copies. See Section 4.1.2.

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. Figure 3.1 shows the set of DDL examples used in this section. We start by defining two views as shown in D1.

```
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 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 region**, **consistency region**, and **completeness region** to refer to cache regions defined for each type, respectively. 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.

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. Furthermore, we also 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 that can be admitted to or evicted from the cache in the MTCache "pull" framework for cache maintenance. We assume that the cache maintenance algorithms materialize, update and evict all rows within such a group together.

<u>**Presence Assumption:**</u> 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 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]. In this paper, we limit our discussion to

only the simplest type of PMVs, namely views with an equality control-table; more general cases are addressed in [ZLG05].

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. Consider a scenario where we receive many queries like E1.3. Suppose the view AuthorCopy contains all the required rows. If we compute the query from the view, will the result satisfy the query's consistency requirements or not? 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. In our example, we first create a consistency control-table containing a set of cities and then add a CONSISTENCY constraint to AuthorCopy, as in D3 of Figure 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 PMV 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 PMV 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 selection query but we cannot tell whether the view contains *all* required rows. If we want to answer queries with predicate P on

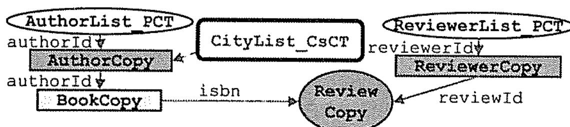

Figure 3.2: Cache schema example

columns other than the control-keys, the cache must guarantee that all rows defined by P appear in the cache. Completeness constraints can be specified in terms of a completeness control-table.

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 Figure 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 in cache management. The cache admin can specify: 1) which subset of columns should be complete; 2) whether 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 many 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 define BookCopy to be controlled by AuthorCopy, we only need to declare AuthorCopy to be a presence control-table by a PRESENCE constraint in the definition of BookCopy, as in D5 of Figure 3.1.

If a presence control-table is not controlled by another presence control-table, 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>**. If there is an edge $V_i \xrightarrow{K_{i,j}, K_{i,j}'} V_j$, then 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 Figure 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 first property: circle for view-level consistent views and rectangle (default) for all others. We use colors to denote the second property: gray if a view is consistency-wise correlated to its parents, red (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, which means that if we add a new author to AuthorCopy, we always bring in all of the author's books. The books for 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 Figure 3.1 . A node with a ROOT consistency constraint is colored *gray*; it cannot have its own consistency or completeness control-tables, as stated in Rule 2 in Figure 5.1.

For a gray node V, we call its closest red ancestor its **consistency root.** For any of V's cache regions U_j, if U_j is controlled by a presence control-key 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.**

Figure 3.2 illustrates a cache schema example.

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, called *safety*, that allows us to optimize pull-based cache maintenance, and summarize the gains it achieves in Theorem 4.2. In the course of the discussion, we also introduce the concept of *ad-hoc* currency regions, which is useful 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 back-end or multiple queries wrapped in a transaction. As an example, suppose AuthorCopy, introduced in Section 3, does not have any children in 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 Figure 4.1 to the back-end 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. Let **K** be the set of authors in AuthorCopy that are from Madison, the consistency query in Figure 4.1 is sent to the back-end 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 Figure 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 Figure 4.2.)

Lemma 4.1: For any cache region U <V, K, k> in the cache, the results retrieved from the back-end server using the above presence, consistency and completeness queries not only keeps U's cache constraints, but also keeps the presence constraints for the presence regions in V that U overlaps. □

Proof: This directly follows from the presence, consistency and completeness queries. □

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 as follows:

1) AC(U) = {U}
2) AC(U) = AC(U) ∪ {U_i | for some 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 any 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). We use **Max(Ω)** to 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) ∪ 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 red node.
2) When we refresh a region U, we follow the following steps:

Step 1: Retrieve all regions in MaxAC(U) by sending remote queries accordingly, let the result be denoted by newTupleSet.

Step 2: Delete the old rows covered by AC(U) or newTupleSet, then insert newTupleSet into the corresponding views. □

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,

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

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. □

Proof: Let Ω = AC(U), maxSet=MaxAC(U), newTupleSet be the tuple set retrieved for maxSet.

1) For any cache region X <V, K, k> in Ω, let V' be the refreshed instance of V, D be the set of rows for V in newRowSet, $X = \delta_{K=k}(V)$, $X' = \delta_{K=k}(V')$, and $X'' = \delta_{K=k}(D)$.

We first prove $X' = X''$. This is obvious from step2 in the maintenance rule, since all the rows in X are deleted and all the rows in X'' are inserted into V'.

Case 1: X is in maxSet. Directly from lemma 4.1.

Case 2: X is in (Ω-maxSet). Then there is a region Y in maxSet, such that $X \subseteq Y$.

Case 2.1: If X is a present region, then directly from lemma 4.1. Otherwise,

Case 2.2: Y has an equal or stronger constraint than X. Since Y observes its constraint (from Case 1), it follows from lemma 2.1, 2.2, 2.3 that so does X.

Case 3: X is not in $\Delta \cup \Omega$. We prove that $X' = X$. This is so because from the maintenance rule, those rows in U are not touched by the refresh transaction.

2) It directly follows from 1).

3) It is obvious if U is the only element in Ω. Otherwise, prove by constructing counterexamples from AuthorCopy. In AuthorCopy, suppose there is a present control table on authorId with authorIds 1 and 2; there are two tuples: t1 = <1, Rose, Female, Madison, WI>, t2 = <2, Mary, Female, Seattle, WA>. Suppose we want to refresh t1 after an update that touched every row in Authors in the master database.

Prove by contradiction. Suppose there exists X in maxSet that should not being refreshed.

Case 1: There exists Y in maxSet, such that $X \subseteq Y$. Due to the definition of the maxSet, X must be a complete region and Y a consistent region.

In AuthorCopy, suppose it has a complete region defined on city with value Madison; a consistency region defined on state with value WI. If a new author from Madison has been added in the master database, if we only refresh the consistent region by WI, only t1 will be refreshed, and after refresh, the completeness constraint on Madison is no longer preserved.

Case 2: There exists a cache region Y in maxSet, s.t. X overlaps with Y. In AuthorCopy, suppose it has two consistent regions on WI and female respectively. If we only refresh the first one, only t1 will be refreshed, and after refresh, the consistency constraint on the latter is no longer preserved. □

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 on this state 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. A general maintenance algorithm is sketched below.

```
Maintenance Algorithm:
INPUT: a cache region U from a red node
{
  Ω ←{U};
  While (TRUE)
 {
 Ω ← AC(Ω);
 maxSet ← MaxAC(Ω);
 oldRowSet =    ∪          Ui //the instance set
             Ui∈ max Set .
 NewRowSet = retrieve(maxSet);
 Δ = AC(NewRowSet);
 If (Δ⊆Ω) break;
 Ω = Δ ∪Ω
}
apply(oldRowSet, newRowSet);
```

Function retrieve(Ω) retrieves rows from the master database by sending a series of remote queries accordingly for each group in Ω.

Procedure apply() refreshes the cache according to step 2 in the second part of the Maintenance Rule.

```
Procedure Apply (S, D)
Input: S - source row set, D - new row set
Algorithm:
  for (each view Vi involved)
  {
     Let the set of rows in S that
             belongs to Vi be Si;
     Let the set of rows in D that
             belongs to Vi be Di;
     Let dkey = Πkey(Di);
     Delete Si from Vi;
     Delete rows in Vi whose keys appear in
             dkey;
     Insert Di into Vi.
  }
```

Given a region U in a red PMV 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 the correspondingly high cost of refreshing the cache motivate the definition of *safe* PMVs 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.

Defn: **(Ad-hoc region)** An ad-hoc cache region consists of a union of one or more regions (which might be from different views) that are mutually consistent. □

Such "ad-hoc" consistency information is made known to the query processor by associating the region id of the ad-hoc region with each region it contains.

4.1.2 Keeping Track of Currency

In order to judge if cached data is fresh enough for a given query, we need to keep track of its currency. It is straightforward and we discuss it only briefly. [GLRG04] used a push model for cache maintenance, and relied on a heartbeat mechanism for this purpose. To track currency when using the pull model, we keep a timestamp for every cache region. When a cache region is refreshed, we also retrieve and record the transaction timestamp of the refresh query. Assuming that a transaction timestamp is unique, in implementation we simply use the timestamp as region id. Thus, if the timestamp (rid) for a cache region is T and the current time is t, since all updates until T are reflected in the result of the refresh query, the region is from a database snapshot no older than $(t - T)$.

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 regions in V defined by presence control-key values. 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 red PMV V that doesn't have any children, given any cache region U in V, the partially ordered set AC(U) is a tree. □

Proof: (by contradiction) Suppose there is a group X in AC(U), such that X has two parents Y and Z. Then $Y \cap Z \neq \emptyset$. From the safe definition, either $Y \subseteq Z$, or $Z \subseteq Y$. Therefore they cannot both be X's parents. □

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

Theorem 4.2: Consider a red 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) The calculation of AC(U) can be done 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. □

Proof:

1) For any safe gray node V', given the subset of PCK values **K** that is in AC(U) from its parent, we need to put in AC(U) the set of cache regions Δ determined by **K** in V'. Δ is the exact set of cache regions in V' that need to be put into AC(U), because from the definition of a safe view, Δ doesn't overlap or contained by any consistent or complete region defined on V', nor does it overlap or contained by the rest of the present CRs in V'. Further, adding Δ to AC(U) doesn't result in adding additional cache regions from its parent, because of the first condition of the definition of safe.

2) From 1), the descendents of V don't affect AC(U) on V. Thus, let Ω = AC(U), from Lemma 4.2, Ω is a tree. Let Γ be empty, we check the tree recursively top down from the root, let it be Y. If a node X is a complete region, then we add it to Γ; Otherwise, we do the checking on each child of X. If Y is not in Γ, add it to Γ.

We prove that Γ = MaxAC(U). If Y is a complete or a present region, we are done. Otherwise, let д, в be the set of all consistent regions and complete regions in Ω respectively. {Y} = Max (Ω- в), since it is the root of the tree. Now we prove Γ -{Y} = Max(Ω- д) by contradiction. Suppose there is a complete region Z in Ω, such that Γ -{Y} doesn't cover Z. Then Z doesn't have any ancestor that is a complete region. Then from the algorithm, Z must be visited and put into Γ -{Y}, contradicting the assumption.

Further, the cleaning step doesn't eliminate Y, since it is the root. Thus Γ = MaxAC(U). □

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 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 have a closer look at the AuthorCopy and BookCopy example defined in Section 3. Suppose a book can have multiple authors. If BookCopy is 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 colored cache DAG <**W**, **E**>, we construct the corresponding **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) Given a cache DAG <**W**, **E**>, a PMV V in **W** has the **shared-row problem** if there exists a cache instance DAG **I** containing a row in V with more than one parent. □

There are two cases where a PMV V has the shared-row problem. The first case is as follows:

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

Proof: (by constructing an instance DAG). Suppose V has two PCTs T1 and T2 on attributes A and B respectively. Suppose values a1 and b1 are in T1 and T2 respectively. For a row t in V, if t.A = a1, t.B = b1, then t has two parents: a1 and b1. Thus V has the shared-row problem. □

In this case, we can only eliminate the potential overlap of regions defined by different presence control-tables if V is view-

level consistent. Considering the second condition in the definition of safe, we enforce Rule 3 in Figure 5.1.

The second case where a PMV has the shared-row problem is identified next. For this case, we enforce Rule 4 in Figure 5.1.

Lemma 5.2: Given a cache schema <**W**, **E**>, for any PMV V in a tree, let the parent of V be V_1, then V has the shared-row problem if and only if the presence control-key K in V_1 for V is not a key in V_1. □

Proof: (sufficiency) Since K is not a key for V_1, there exists an instance of V_1, such that there are two rows t1 and t2 in V_1, such that t1. K = t2. K. Then for a row t in V, s.t. t.K=t1.K, both t1 and t2 are t's parents.
(necessity) Because V has the shared-row problem, there is an instance of V, such that a row t in V has two parents, t1 and t2 in V_1. Since t1.K = t2.K= t.K, K is not a key for V_1. □

5.2 Control-table Hierarchy

For a red PMV V in the cache, if it has some consistency or completeness control-tables beyond those implicit in the Presence Assumption, then it might have overlapping regions. In our running example, suppose BookCopy is a red rectangle; an author may have more than one publishers. If there is a consistency control-table on publisherId, then BookCopy may have overlapping regions. As an example, author 1 has books 1 and 2, author 2 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 author 1, and {book 1, book 3} by publisher A.

Defn: (Compatible Control-tables) For a PMV V with one presence control-table 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 Figure 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 PMV 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 PMV is unsafe. □

Proof: (Sufficiency) by contradiction. Suppose there exists a PMV V that is not safe. There are two cases:

Case 1: There exists a pair of cache regions U1 and U2 in V, s.t. U1 and U2 overlap.

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

Figure 5.1: Cache schema design rules

This violates Rule 5.

Case 2: V is grey. Let Ω denote the set of cache regions in V defined by its presence control-key values. Again, there are two cases:

Case 2.1: There are U1 and U2 in Ω, such that U1 and U2 overlap.

This implies that V has shared-row problem. Then it violates rule 3 or 4.

Case 2.2: There are U1 and U2 in Ω, and U3 in V, such that U1 and U2 are contained in U3.
This implies that V has its own consistency control-tables, which violates rule 2.

(Necessity) We use variations of the cache schema in Fig 3.1 as counter examples in a proof by contradiction.

Case 1: Rule 1 is violated. Then <W, E> violates the defn of cache schema.

Case 2: Rule 2 is violated.

Suppose BookCopy is required to be consistent by type; author a1 has books b1 and b2; a2 has a book b3; and b1, b2, b3 are all of type paperback. Then BookCopy is not safe because cache regions {b1, b2} (by a1), {b3} (by a2) are contained in the one defined by paperback type.

Case 3: Rule 3 is violated.

Suppose ReviewsCopy is a rectangle or gray. If it is a rectangle, suppose book b1 has two reviews r1, and r2, from reviewers x and y, respectively; x wrote reviews r1 and r3. Since cache regions {r1, r2} (by b1) and {r1, r3} (by x) overlap, ReviewsCopy is not safe.

Next, if ReviewsCopy is a circle, suppose author a1 has books b1 and b2; author a2 has a book b3; books b2, b3 have reviews r2, r3, respectively. Since cache regions {b1, b2} (by a1) and {b2, b3} (by correlation with ReviewsCopy), BookCopy is not safe.

Case 4: Rule 4 is violated.

Suppose a book can have multiple authors and BookCopy is gray. Suppose AuthorsCopy is consistent by city; author a1 has books b1 and b2; author a2 has books b1 and b3; author a1 and a3 are from WI, a2 is from WA.

First, suppose BookCopy is a rectangle. Since cache regions {b1, b2} (by a1), {b1, b3} (by a2) overlap, BookCopy is not safe.

Second, suppose BookCopy is a circle. Since cache regions {a1, a3} (by WI), and {a1, a2} (by consistency correlation with BookCopy) overlap, AuthorsCopy is not safe.

Case 5: Rule 5 is violated.
Suppose ReviewersCopy is required to be consistent both by gender and by city; reviewers x and y are from WI, z is from WA; x and z are male, while y is female. Since cache regions: {x, y} (by WI), {x, z} (by male) overlap, ReviewsCopy is not safe. □

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. Theorems 6.1-6.3 guarantee the correctness of our algorithms.

Different from SGLRG04], the algorithms developed in this paper are more general; they support finer granularity C&C checking. In [GLRG04], consistency checking was done at optimization

time and currency checking at run-time, because view level cache region information is stable and available at query compile time, while currency information is only available at runtime. In contrast, in this paper we still perform as much as possible of the consistency checking at optimization time but some checking may have to be delayed to run-time. If we are using a PMV with partial consistency guarantees, we don't know at optimization time which actual groups will be consistent at run time. Furthermore, 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 in the following form.

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 each $\mathbf{S}_i$ 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 $\mathbf{S}_i$, $\mathbf{G}_i$ specifies a grouping key and $\mathbf{K}_i$ specifies 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 that will be checked at run time. Note that the default value for each field is the strongest constraint.

To enable efficient reasoning, we union together all constraints from individual currency clauses into a single constraint, and convert the result into an equivalent or stricter normalized form with no redundant requirements.

Defn: (Normalized C&C constraint) A C&C constraint CCr = $\{<b_1, \mathbf{K}_1, \mathbf{S}_1, \mathbf{G}_1>, ..., <b_n, \mathbf{K}_n, \mathbf{S}_n, \mathbf{G}_n>\}$ is in normalized form if all input operands (in the sets $\mathbf{S}_i$) are base tables and the input operand sets $\mathbf{S}_1, ..., \mathbf{S}_n$ are all non-overlapping. □

We briefly sketch an algorithm for transforming a set of constraints into normalized form. First, recursively expand all references to views into references to base tables. Next, repeatedly merge any two tuples that have one or more input operands in common using the following rule.

Normalization Rule: Given $CCr_1 = \{<b_1, \mathbf{K}_1, \mathbf{S}_1, \mathbf{G}_1>\}$ and $CCr_2 = \{<b_2, \mathbf{K}_2, \mathbf{S}_2, \mathbf{G}_2>\}$, $\mathbf{S}_1 \cap \mathbf{S}_2 \neq \emptyset$, replace the two constraints by CCr = $\{<b, \mathbf{K}, \mathbf{S}, \mathbf{G}>\}$, where $b = \min(b_1, b_2)$, and $\mathbf{S} = \mathbf{S}_1 \cup \mathbf{S}_2$. Given a set of functional dependencies (FDs) F over the query result relation Y, let $\mathbf{G}_i^+$ be the attribute closure of $\mathbf{G}_i$ w.r.t. F, where i = 1, 2. Then $\mathbf{G} = \mathbf{G}_1^+ \cap \mathbf{G}_2^+$. Let $\mathbf{K}_i^+ = \Pi_G \sigma_{Gi=ki}(Y)$, i = 1, 2. Then $\mathbf{K} = \mathbf{K}_1^+ \cup \mathbf{K}_2^+$. □

Given a set of FDs over the base relations, and the equivalence classes induced by a query, we can infer the set of FDs over the query result relation. For example, for Q2, let CCr_1 = {<10, Ø, {Authors, Books}, {city}>}, CCr_2 = {<5, Ø, { Books}, {isbn}>}. CCr_1 requires that if we group the query result by city, then within each group, all the rows have to be consistent. CCr_2 requires that if we group the result by isbn, then each book row has to be consistent. From the key constraints in Authors and Books, together with the join condition in Q2, we know that isbn is a key for the final relation. Thus CCr = {<5, Ø, {Authors, Books}, {city}>}. If an instance satisfies CCr, then it must satisfy both CCr_1 and CCr_2, and vice versa.

In what follows, we formally define implication and equivalence between any two CCrs, and prove that when K1 and K2 are set to default, then the outcome of the normalization rule CCr is equivalent to the inputs $CCr_1 \cup CCr_2$ w.r.t. F. Further, we prove that not knowing all FDs doesn't affect the correctness of the rule.

Defn: (Implication, Equivalence) Given two C&C constraints CCr_1 and CCr_2, a cache schema Λ, and a set of FDs F over Λ, we say that CCr_1 **implies** CCr_2 w.r.t Λ and F, if every instance of Λ that satisfies F and CCr_1 also satisfies CCr_2. If CCr_1 implies C2 w.r.t Λ and F and CCr_2 implies C1 w.r.t Λ and F, then CCr_1 and CCr_2 are **equivalent** w.r.t Λ and F. □

Lemma 6.1: For any CCr = $\{<b, \mathbf{K}, \mathbf{S}, \mathbf{G}>\}$, any instance of Λ, the consistency constraint in t can be satisfied w.r.t. Λ and F, iff the grouping key $\mathbf{G'}$ of the cache region partitioning on S in Λ is a subset of $\mathbf{G}^+$ w.r.t. Λ and F. □

Proof: Sufficiency is obvious. Now we prove necessity. Since each group by grouping key $\mathbf{G}$ belongs to one group by grouping key $\mathbf{G'}$, $\mathbf{G}$ functionally determines $\mathbf{G'}$. Thus $\mathbf{G'} \subseteq \mathbf{G}^+$. □

Theorem 6.1: If $\mathbf{K}_1$ and $\mathbf{K}_2$ are set to default, then the output of the Normalization Rule CCr is equivalent to its input $CCr_1 \cup CCr_2$ w.r.t. Λ and F. □

Proof: Given any instance of Λ that satisfies {CCr} w.r.t. to F, from Lemma6.1, the grouping key of its cache region partitioning is a subset of $\mathbf{G}^+$. Since $\mathbf{G} \subseteq \mathbf{G}_i^+$, i = 1, 2, $\mathbf{G}^+ \subseteq \mathbf{G}_i^+$, the consistency constraints in ($CCr \cup CCr_2$} are satisfied. Further, since the consistency portioning satisfies currency constraint b, and b = min (b_1, b_2), b_1 and b_2 are also satisfied.

From Lemma 6.1, it follows that for any instance that satisfies both t_1 and t_2 w.r.t. F, the grouping key of its cache region partitioning has to be a subset of $\mathbf{G}$. Thus, it also satisfies t. Since it satisfies b_1 and b_2, and $b = \min(b_1, b_2)$, it also satisfies b. □

Theorem 6.2: Suppose FDs over a cache schema Λ: $F+ \subset F'+$. The output of the Normalization Rule {CCr} w.r.t. F implies its input $CCr_1 \cup CCr_2$ w.r.t. Λ and F'. □

Proof: Let $\mathbf{G} = \mathbf{G}_1^+ \cap \mathbf{G}_2^+$ w.r.t. F, $\mathbf{G'} = \mathbf{G}_1^+ \cap \mathbf{G}_2^+$ w.r.t. F'. Then $\mathbf{G} \subseteq \mathbf{G'}$. Thus for any instance of Λ that satisfies CCr, since $\mathbf{K} = \mathbf{K}_1^+ \cup \mathbf{K}_2^+$ w.r.t. F, from Lemma 6.1, it satisfies $CCr_1 \cup CCr_2$. □

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 to the plan, which check at runtime whether the guarantee holds for the groups actually used.

We use a transformation-based optimizer. Conceptually, optimization proceeds in two phases: an exploration phase and an optimization phase. The exploration phase generates new logical expressions; the optimization phase 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. There are many plan properties

but we'll illustrate the idea with the sort property. A merge join operator requires that its inputs be sorted on the join columns. To ensure this, the merge join passes down to its input a required sort property. In essence, the merge join is saying: "Find me the cheapest plan for this input that produces a result sorted on these columns." Every physical plan includes a delivered sort property that specifies if the result will be sorted and, if so, on what columns and in what order. Any plan whose delivered properties do not satisfy the required properties is discarded.

To make use of the plan property mechanism for consistency checking, we must be able to perform the following three tasks: i) transform the query's consistency constraints into required consistency properties; ii) given a physical plan, derive its delivered consistency properties from the properties of the local views it refers to; iii) 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 the previous section.

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. In what follows, we sketch the algorithm for computing a plan's delivered consistency properties but skip the detailed algorithm due to space constraints.

Delivered plan properties are computed bottom-up for each physical operator, in terms of the properties of its inputs, according to the **Delivered-Plan Algorithm** described below, which treats the physical operators accordingly as four categories: i) leaves of the plan tree (e.g., tables or materialized views), ii) single-input operators, iii) joins, and iv) SwitchUnion.

Delivered-Plan Algorithm (sketch)

The *leaves of a plan tree* are table, materialized view, or index scan operators, possibly with a range predicate. If the input operand is a local view, we return the ids of the view's input tables in **S**, not the id of the view, since consistency properties always refer to base tables. If the whole view is consistent, we simply return the id of its cache region; otherwise, we return the set of grouping keys of its consistency root, and a flag, say –1, in the region id field to indicate row-level granularity. For a remote table or view, we do the same, except we assume it is consistent with a special region id, say, 0.

All *operators with a single relational input*, such as filter, project, aggregate and sort do not affect the delivered consistency property and simply copy the property from their relational input.

Join operators combine two input streams into a single output stream. We union the input consistency properties and merge property tuples that are in the same cache region. Formally, given two delivered C&C property tuples $CPd_1 = \{<R_1, \mathbf{S}_1, \mathbf{\Omega}_1>\}$ and $CPd_2 = \{<R_2, \mathbf{S}_2, \mathbf{\Omega}_2>\}$, we merge them if either of the following conditions is true:

1) If the input operands are from the same cache region, i.e., $R_1 = R_2 \geq 0$, then we merge the tables, i.e., we replace CPd_1 and CPd_2 by $CPd = \{<R_1, \mathbf{S}, \emptyset>\}$, where $\mathbf{S} = \mathbf{S}_1 \cup \mathbf{S}_2$.
2) If the input operands are grouped into cache regions by the same keys (for the same root), i.e., $\mathbf{\Omega}_1 = \mathbf{\Omega}_2$, they are group-wise consistent so we merge them into $CPd = \{< -1, \mathbf{S}, \mathbf{\Omega}_1>\}$ where $\mathbf{S} = \mathbf{S}_1 \cup \mathbf{S}_2$.

A *SwitchUnion* operator has multiple input streams but it does not combine them in any way; it simply selects one of the streams. Thus, the output consistency property is the strongest consistency property implied by every input. In our context a *SwitchUnion* operator has a local and a remote branch. We output the properties of the local branch. □

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 such that $\mathbf{S}_1 \cap \mathbf{S}_2 \neq \emptyset$ and one of the following conditions holds: i) $R_1 \neq R_2$, or ii) $\mathbf{\Omega}_1 \neq \mathbf{\Omega}_2$. □

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 (the second condition).

Secondly, we can verify that a complete plan satisfies the constraint by checking that 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, if and only if CPd is not conflicting and, for each tuple $<b_r, \mathbf{K}_r, \mathbf{S}_r, \mathbf{G}_r>$ in CCr, there exists a tuple $<R_d, \mathbf{S}_d, \mathbf{\Omega}_d>$ in CPd such that $\mathbf{S}_r \subseteq \mathbf{S}_d$, and one of the following conditions holds: i) $\mathbf{\Omega}_d = \emptyset$, or ii) let $\mathbf{G}_r^+$ be the attribute closure w.r.t. F. There exists a $\mathbf{G}_d \in \mathbf{\Omega}_d$ such that $\mathbf{G}_d \subseteq \mathbf{G}_r^+$. □

For query Q2, suppose we have CCr = {<5, Ø, {Authors, Books}, {isbn}>}, and that the cache schema is the one in Figure 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$\in$ $\{isbn\}^+$. CPd satisfies CCr.

Not knowing all FDs doesn't affect the correctness of the satisfaction rule, it only potentially produces false negatives:

Theorem 6.3: For any two sets of functional dependencies F and F' over the cache schema Σ, where $F^+ \subseteq F'^+$, if a delivered consistency property CPd satisfies a required CCr w.r.t. F, then it satisfies CCr w.r.t. F'. □

Proof: Let $\mathbf{G}_r^+$ be the attribute closure of $\mathbf{G}_r$ w.r.t. F^+, $\mathbf{G}_r'^+$ be the attribute closure of $\mathbf{G}_r$ w.r.t. F'^+, then $\mathbf{G}_r^+ \subseteq \mathbf{G}_r'^+$. □

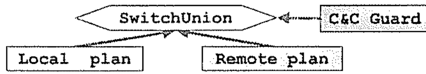

Figure 6.1: SwitchUnion with a C&C guard

Theorem 6.4: Assuming runtime checking is correct, with the Delivered-Plan Algorithm, for any plan of which CPd satisfies CCr w.r.t. a cache schema Σ and functional dependencies F, no matter which data sources are used at execution time, CCr will be satisfied w.r.t F. □

Proof: Let the set of C&C properties of the sources be CPd = {< R_{di}, S_{di}, $\boldsymbol{\Omega}_{di}$ >}. Let the output of the Delivered-Plan Algorithm be CPd'.

Case 1: There are no SwitchUnion operators in the plan.

Since operators with a single relational input simply pass the input property; while join operators simply merge the input properties with the same cache region, we have CPd = CPd'.

Case 2: There are some SwitchUnions used as C&C guards.

In this case, for each SWU, there are two types of checking: fullness checking and currency checking. So the branch actually used satisfies the fullness and currency constraint.

The difference between CPd and CPd' is that in CPd, for a local source with property CPd_i = {< R_{di}, S_{di}, $\boldsymbol{\Omega}_{di}$>} guarded with a SWU, we have either CPd_i or CPd_i' = {<0, S_{di}, Ø>}, depending on whether the local branch or the remote branch is used during execution.

For any tuple r = <b_r, $\mathbf{K}_r$, S_r, G_r> in CCr, since CPd' satisfies CCr, there exists a row t = <R_d, S_d, $\boldsymbol{\Omega}_d$ >, such that, $S_r \subseteq S_d$, and one of the following conditions holds: i) $\boldsymbol{\Omega}_d$ = Ø, or ii) let $\mathbf{G}_r$+ be the attribute closure w.r.t. F. There exists a $\mathbf{G}_d \in \boldsymbol{\Omega}_d$ such that $\mathbf{G}_d \subseteq \mathbf{G}_r^+$.

If t is merged from sources that don't have a swu, then it also appears in CPd, otherwise, w/o loss of generality, we can assume it comes from two local resources with swu operators and with property t_1 = < R_{d1}, S_{d1}, $\boldsymbol{\Omega}_{d1}$> and t_2 = < R_{d2}, S_{d2}, $\boldsymbol{\Omega}_{d2}$>.

Trivial case: If $S_r \subseteq S_{d1}$(or S_{d2}), then r is satisfied by t_1 (or t_2) in CPd.

Otherwise, we claim that for any cache instance, either both local branches are used or both remote branches are used. Thus if CPd' satisfies CCr, then if we plug in CPd the property of the data sources actually used, CPd also satisfies CCr.

Case 1: R>0. Since both local resources belong to the same cache region, they have the same currency, so does the currency checking result.

Case 2: R= -1. Since the two resources are controlled by the same set of consistency control-keys, again, the C&C checking results are the same. □

While a plan is being constructed, bottom-up, we want to stop as soon as it is possible to tell that the plan cannot deliver the consistency required by the query. Unfortunately, the consistency satisfaction rule cannot be used for this purpose as soon as a new root operator is added to a plan; a check may fail simply because the partial plan does not include all inputs covered by the required consistency property. Therefore we develop a *violation rule*.

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

Theorem 6.5: Using the Delivered-Plan Algorithm, if a partial plan A violates the required consistency property CCr w.r.t. a cache schema Σ and functional dependencies F, then no plan that includes A as a branch can satisfy CCr w.r.t. Σ and F . □

Proof: This is true because from the algorithm, for any tuple < R_d, S_d, $\boldsymbol{\Omega}_d$ > in the delivered plan property of P, there is a tuple < R_d, S_d', $\boldsymbol{\Omega}_d$ > in the delivered plan property of any plan that includes P as a branch, where $S_d \subseteq S_d$'. □

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 required C&C constraints and switch between using the local view and retrieving the data from the back-end server. Such run-time decision-making is built in a plan by using a *SwitchUnion* operator. A SwitchUnion operator has multiple input streams but it does not combine them in any way; it simply selects one of the streams according to the outcome of a selector expression.

All local data is defined as materialized views and logical plans making use of a local view are always created through view matching [LGGZ04, LGZ04]. Consider an (logical) expression E and a matching view V from which E can be computed. If there is no C&C checking required on the input tables of E, view matching [GL01] produces a "normal" substitute consisting of, at most, a select, a project and a group-by on top of V. With C&C checking, view matching produces a substitute consisting of a SwitchUnion on top, shown in Figure 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 the "normal" substitute mentioned earlier 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 branch is chosen.

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

Currency bound checking: If the required lowest currency bound on the input tables of E is B, the optimizer generates a currency guard that checks if any required region is too old for the query. Given a control-table CT on control-key CK, a set of probing values **K** on CK, recall that the timestamp is recorded in the rid column of each control-table (Section 4.1.2), the check is:

```
NOT EXIST (SELECT 1 FROM CT
   WHERE CK IN K AND rid < getdate()-B)
```

7. PERFORMANCE STUDY

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

We used a single cache DBMS and a back-end server. The back-end server hosted a TPCD database with scale factor 1.0

(about 1GB). The experiments reported here used only the Customers and Orders tables. 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 DBMS had a copy of each table, CustCopy and OrderCopy, with the same indexes. The control-table settings and queries used are shown in Figure 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 rid 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 back-end ran on an AMD Athlon MP Processor 1800+ box with 2GB RAM. Both machines ran Windows 2000 and were connected in a local area network.

7.1 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 several types 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 back-end. We measure the overhead for both scenarios.

7.1.1 Single Table Case

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

Condition A: Is each required consistency group equal to or contained in a region defined by the presence control-table?

If A is true, 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 region defined by a consistency control-table?

If B is true, we check C, otherwise we check D.

Condition C: Is the consistency guarantee full?

If C is true, 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, Qa does not include an equality predicate on c_nationkey. We have to first calculate which nations are in the results, 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. Such checking (e.g., 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, the predicate on nation is included in the query as a redundant predicate, which allows us to simply check if each required nation is in C_CsCT (A12). It eliminates the need to examine the data before consistency checking.

Condition D: Each required consistency group can be covered by a collection of cache regions.

If D is true, 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, the ad-hoc checking needs to check each group (S12).

Experiment 1 is designed to measure the overhead of the type of consistency guards supported in our current framework. We choose to support only run-time consistency guards that i) do not require touching the data in a PMV; ii) only require probing a single control-table. To be specific, we only support the guards shown in A12 and S11. We fixed the guards and measured the overhead for three different queries: 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 (less than 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

```
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

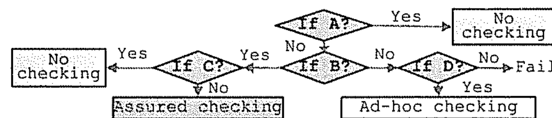

Figure 7.2: Generating consistency guard

```
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: Consistency guard examples

Cost	Local			Remote		
	Qa	Qb	Qc	Qa	Qb	Qc
ms	.078	.08	1.17	.01	.19	1.13
%	16.5	14.0	<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

(<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 PMV (as in A11a and 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%).

7.1.2 Multi-Table Case

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

Condition E : Do they have the same consistency root?

If E is true, 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 a) all the customers are from the same region; and b) 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.

It is worth pointing out that in all experiments, even for complicated consistency guards, the overhead of remote execution is relatively small (<10% for single-table case, <25% for multi-table case). It raises an interesting point: even if a guard is less likely to be successful, it might still be preferable to do the check than simply use a remote plan. Thus the cost-model should bias in favor of plans with consistency checking instead of remote plans.

7.2 Success Rate of Ad-hoc Checking

Intuitively, everything else being equal, the more relaxed the currency requirements are, the more queries can be computed locally. Although less obvious, this is also true for consistency constraints.

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

Assuming all rows in $custSet are in the cache, a dynamic plan for Qa will switch between either CustCopy and a remote query, depending on the outcome of the consistency guard. If there is only one customer in $custSet, by default the result is consistent. At the other extreme, if $custSet contains 1000 customers, they are not likely to be consistent. When the number of customers in $custSet increases, the likelihood of the result being consistent decreases. Suppose there are N rows in CustCopy, divided into M cache regions. We assume that the regions are the same size and each row in $custSet is independently and randomly chosen from CustCopy. Let the size of $custSet be x, where $x \leq N$. The result is consistent only when all the chosen rows are from the same cache region. Thus, the probability of an ad-hoc consistency check being successful is $P(consistent) = (1/M)^{X-1}$.

As one would expect and this formula clearly shows, the probability of success decreases rapidly as the number of consistency groups and/or number of required rows increase.

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 etc.. Some of them take a maintenance-centric approach [ABG88, GN95, SK97], where queries are not allowed to express their individual data quality requirements. Some authors 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 work 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 two 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.

10. REFERENCES

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

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

[AJL+02] J. Anton, L. Jacobs, X. Liu, J. Parker, Z. Zeng, T. Zhong. Web Caching for Database Applications with Oracle Web Cache. In *SIGMOD*, 2002.

[APTP03] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A Dynamic Data Cache for Web Applications. In *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. In *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. In *VLDB*, 2001.

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

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

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

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

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

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

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

[LGGZ04] P. Larson, J. Goldstein, H. Guo, J. Zhou. MTCache: Mid-tier Database Cache in SQL Server. In *Data Engineering Bulletin*, 2004.

[LGZ04] P. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent Mid-Tier Database Caching In Sql Server. In *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. In *SIGMOD*, 2002.

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

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

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

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

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

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

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

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

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

[ZLG05] J. Zhou, P. Larson, J. Goldstein. Partially Materialized Views, *submitted to this conference.*