Deterministic Prefetching for Container-Managed Persistence

Ahmet S. Bilgin[1], Rada Y. Chirkova[1], Munindar P. Singh[1], Timo J. Salo[2]

1: North Carolina State University, Raleigh, NC 27695, USA
2: IBM, RTP, NC 27703, USA

{asbilgin@ncsu.edu, chirkova@csc.ncsu.edu, singh@ncsu.edu, tjsalo@us.ibm.com}

Abstract

Modern information system architectures place applications in an application server and persistent objects in a relational database. In this setting, we consider the problem of improving application throughput; our proposed solution uses data prefetching to minimize the total data-access time of an application, in a manner that affects neither the application code nor the backend DBMS. Our methodology is based on analyzing and automatically merging SQL queries to produce query sequences with low total response time, in ways that exploit the application's data-access patterns. The proposed approach is independent of the application domain and can be viewed as a component of container-managed persistence that can be implemented in middleware.

This paper describes our proposed framework for using generic data-access patterns in merging queries, to derive query sequences with improved total data-access time. For each guideline that is discovered, we list the rules to determine when the specific guideline is applicable, its key parameters, and the experimental results in terms of the improved data-access time. The approach is evaluated in the context of financial and manufacturing domains, which support the kinds of natural conceptual relationships where this approach is valuable.

This research is supported under NCSU CACC Grant 11019

1 Introduction

Three-tier application architectures place business logic on an application server and the necessary persistent objects on backend relational database management systems (RDBMSs). A major performance problem is that application object models are inherently navigational. Objects have references or relationships to other objects, which applications follow one at time. The navigational characteristics of applications increase the number of roundtrips in three-tier application architectures. These roundtrips result in unacceptable physical disk access and network processing overhead.

In current practice, programmers can spend inordinate amounts of time tuning their applications to reduce the abovementioned overhead; an additional drawback of tuning is that it must be repeated each time the database schema or application logic are modified.

An alternative way to reduce the data-access overhead is to reduce the number of database roundtrips required to fulfill an application's request for stored objects. There are two types of possible approaches: *caching* and *prefetching*. Caching [8, 16, 12, 13] refers to storing recently accessed objects, thereby avoiding unnecessary requests to the database. Prefetching [9, 2, 17, 10, 18] involves fetching data based on a prediction of an application's future requests. Both caching and prefetching can result in significant payoffs in data-access performance [15, 1]. For instance, studies of prefetching have reported improvements of several times by prefetching multiple tuples at a time instead of just the one that is requested [2]. However, to successfully apply prefetching requires determining the *prefetching quantity*, i.e., the number of objects or pages that should be prefetched.

Prefetching techniques are classified into three categories [14]:

- Deterministic prediction techniques use a fixed a strategy. Sequential prefetching creates threads for queries that return large quantities of data sequentially from a single table. List prefetching reads the data according to the index structure, so that data pages do not need to be consecutive. These techniques are used by commercial data servers and by object managers or containers in application servers.

- Object structure-based prefetching techniques predict the access via pointers from objects to other objects, which are mostly used in OODBMSs. In object structure-based prefetching, the middleware object-manager determines what to prefetch in response to navigational access [11, 2]. Here, interrelated objects are modeled as complex objects using the reference and collection attributes. However, interrelated data is modeled via foreign keys and the multiplicity of relations in relational schema.

- Statistical prediction techniques generate probabilistic information about future accesses by analyzing past accesses.

We develop *deterministic prefetching*, an adaptive approach for reducing the number of data-access roundtrips [3, 5]. This approach combines deterministic and object structure-based prefetching techniques for RDBMSs. It provides guidelines that can be used to automatically and iteratively improve data throughput without modifying the application code or the DBMS. Consequently, this approach can be used in middleware in managed object environments. Instead of making (probabilistic) predictions of future queries of an application, we focus on merging (augmenting) the individual queries efficiently to accommodate the result of the subsequent queries under the assumption of perfect knowledge for future queries. We name these augmented queries *prefetch* queries.

The basic primitive step in deterministic prefetching is to process SQL queries as requested by an application to produce a sequence of queries that are sent to the backend DBMS. The typical way to process input queries is to merge into a large query. Whether such merging is appropriate and, if so, which kind of merging is appropriate depends on the queries generated by an application and the schema of the database. The measure of performance we consider here is total response time. Consequently, the goal is to figure out when what strategy would be desirable. What makes a query a suitable candidate for merging with other queries is the central question addressed by this thesis.

The proposed framework comprises interactive query exploration and automatic query analysis based on application behavior, and can be used as a component of autonomic self-tuning data-access systems for data-intensive applications.

Contributions We develop our approach for settings where an application's likely next few queries are known beforehand (e.g., canned interfaces), and the same query is expected to be reissued many times. Application domains that will benefit from our approach include health care, finance, human resources, and OLTP, in general. For these settings, we propose a set of techniques for

- Automatically analyzing an application's data-access patterns.

- Using the results of the analysis and guidelines to prefetch the answers to future queries using sequences of prefetch SQL queries that yield low total response time.

Our results are not meant to be used in individual applications to rewrite queries in a static fashion. Instead, our objective is to create application-independent techniques that middleware would use as the basis for providing an online, adaptive way of implementing prefetching. Thus, our contributions are to autonomic computing, namely to the development of automated systems in managed object environments, with the ability to iteratively analyze data-access patterns and choose appropriate guidelines for merging (or augmenting) application queries, at runtime and with no human intervention. The main feature of our approach is the discovery and use of guidelines for selecting, based on an application's access patterns and additional parameters, efficient ways of merging the application's data requests into prefetch statements.

2 Problem Formulation and Assumptions

Applications usually are structured: they access objects whose relationships are expressed as associations in the schema of the underlying database. Stored data is accessed according to these associations. Many important applications, including those in health care, finance, or human resources, use the same query templates repeatedly. For example, an application can request the due date of a credit-card payment after requesting the balance; or it can request the transactions of the same card in the last billing period. Such commonplace associations and dependencies form a basis for formulating useful application-independent data-access patterns. Moreover, in many applications, the likely next few queries are known beforehand.

We confine ourselves to settings where persistent objects are stored in a RDBMS and where an application's likely upcoming are known beforehand. We consider the problem of reducing the number of data-access roundtrips in furnishing the application's data requests.

Specifically, our objective is to come up with a set of general guidelines and techniques for automatically analyzing the data-access patterns of each given application and constructing a sequence of prefetch queries with low total response time. In simple terms, the key challenge for prefetching is determining the quantity. In our approach, this translates into determining "which queries" to merge and "how" to merge them.

2.1 Prefetch Queries

We assume that applications submit their data requests as unnested SPJ (select-project-join) SQL queries. Each such set of queries can be partitioned into sets of queries; each of the sets in the partition can be combined into one prefetch query and submitted to the DBMS. Our goal is to find the partition S^n of a given query set W of n queries such that it has minimal total response time on the database and brings at least the correct answer. But finding optimal partitions is not tractable. As shown in [4], the number of partitions for a query sequence with n queries increases exponentially with n. Thus it is infeasible to check the response time of all partitions to find the optimal partition.

2.2 Patterns of Query Sequences

Accordingly, we develop a heuristic approach, which proposes guidelines that suggest prefetch query partitions based on an analysis of the pattern correlations in the query sets, the cost of bringing redundant or useless (false) data to the requester, the size of the buffer cache, which accommodates the output tuples, and the cost of the merge operation that is required to process the set of queries together.

Determining the efficient (near-optimal) or optimal prefetch query partitions requires to answer *how* and *when* to merge the individual queries. To answer the both questions, we first classify the queries according to the pattern correlation set they are member of. This classification helps us in determining candidate mergable queries, because merging queries from different sets is either not effective or provides very small improvement. Query sequences in navigational applications exhibit the following three basic different patterns:

Browse pattern, where a sequence of related queries, whose input depends on the outcome of the previous queries, are issued. The previous (outer query) will have an outcome, which consists of more than one tuple. However, the subsequent query or queries can only have input parameter(s), which corresponds to the attribute(s) of only one output tuple from the previous queries, e.g., Q_1, Q_2, and Q_1, Q_7. a_num is an output column for Q_1, and this output is used as input variable for Q_2. The same situation holds for ag_id between Q_1 and Q_7.

Input pattern, where queries have common input parameters, e.g., Q_1 and Q_9 has the same input parameter value for customer id.

Output pattern, where queries have common output parameters that is the queries either have common output variable which is primary or foreign key, or they have an output variable, which references the other, e.g., Q_1 and Q_6, have output variables where *a_num_to* references *a_num*. This pattern is especially observed when the sequence of queries return the same foreign key parameters from different tables for different input parameters;

Example query set In the following query sequence, Q_1 retrieves the account numbers and corresponding agreement ids of a customer with id = 4000. After looking up the answer of Q_1, for specific account numbers a_num_1, a_num_2, and a_num_3, the related information about the transactions on these accounts are retrieved by Q_2, Q_3, and Q_4. Moreover, the product ids, and other accounts which are related to account number a_num_3, are also retrieved by Q_5, and Q_6. Also for agreements ag_id_1 and ag_id_2, the agreement amount is returned. Then, for the same customer, his orders and the items he ordered, are retrieved. For some or all of the items, this customer ordered, the components information is retrieved, and for one of these components ($b\_component\_id_1$), its suppliers and the quantity supplied are also retrieved by Q_13.

Q_1 : select *a_num*, *ag_id* from AccountRole, Account where $ar\_customer\_id$ = 4000 and $ar\_a\_num = a\_num$

Q_2 : select *at_id*, *at_amount* from AccountTransaction where $at\_a\_num = Q_1.a\_num_1$

Q_3 : select *at_id*, *at_amount* from AccountTransaction where $at\_a\_num = Q_1.a\_num_2$

Q_4 : select *at_id*, *at_amount* from AccountTransaction where $at\_a\_num = Q_1.a\_num_3$

Q_5 : select *product_id* from AccountProducts where $ap\_a\_num = Q_1.a\_num_3$

Q_6 : select *a_num_to* from AccountRelations where $a\_num\_from =$ "$Q_1.a\_num_3$"

Q_7 : select *agreement_amount* from Agreement where $ag\_id = Q_1.ag\_id_1$

Q_8 : select *agreement_amount* from Agreement where $ag\_id = Q_1.ag\_id_2$

Q_9 : select *ol_order_id*, *ol_item_id* from Orders, Orderline where $o\_customer\_id$ = 4000 AND $o\_id = ol\_order\_id$

Q_{10} : select *b_component_id* from Bom where $b\_assembly\_id = Q_9.ol\_item\_id_1$

Q_{11} : select *b_component_id* from Bom where $b\_assembly\_id = Q_9.ol\_item\_id_2$

Q_{12} : select *b_component_id* from Bom where $b\_assembly\_id = Q_9.ol\_item\_id_3$

Q_{13} : select *supplier_id*, *qty_demanded* from *Supplier_component* where $comp\_id = Q_{12}.b\_component\_id_1$

In the above query sequence example, Q_1 forms a browse pattern on the same input parameter and same output parameters but with different values for multiple queries, e.g., Q_1 and the query set $\{Q_2, Q_3, Q_4\}$, Q_1 and the query set $\{Q_7, Q_8\}$, Q_1 and Q_5, or Q_1 and Q_6 . Bowman and Salem [7] define a *nested pattern* as the existence of multiple browse patterns on the same input parameters, but do not give detailed formalization of this pattern, e.g., Q_1 has browse pattern correlation type with all the queries from Q_2 to Q_8, but all these correlations are different, because of the different output parameters of the different queries. We introduce and formalize the *browse* pattern as a generalization of the nested pattern in Section 4. The browse pattern is especially popular in web-based applications where a look-up query is issued first and the detailed information about one of the objects returned by this look-up query, is requested in the subsequent queries. The query sets $\{Q_1, Q_9\}$ or $\{Q_4, Q_5, Q_6\}$ is an example for the input pattern because both queries have the common *customer_id* or *a_num* values as an input parameter value. The required condition to have input pattern correlation for the given queries is to have at least one common input parameter among the queries and not to have browse pattern correlations among these queries. Our main focus in this project is the query sequences that has browse and input pattern correlations. These patterns are easier to handle and provides enough opportunity to merge the queries. The output pattern is also an important query pattern for business applications, but semantic caching or predicate-based caching mechanisms [13, 8], where the database is accessed only for the remainder of the subsequent query, already deals with this pattern correlation type.

There are three main questions that is required to be answered to determine the size and the structure of the prefetched data:

- How much to prefetch?
- How deep to prefetch?
- In what direction to prefetch?

Query pattern correlations are useful in terms of providing hints to answer these questions. The input pattern provides us information about how much more data for the given object can be prefetched efficiently through breadth-first traversal of the tables in the database; and the browse pattern provides us information about how deep we can prefetch data starting from a root object through depth-first traversal.

In a query sequence, any mixture of the above patterns can be also observed among subsequent queries. A query can be a member of many patterns with different query sets, e.g., Q_4 has the browse pattern with Q_1 and the input pattern with Q_5. Detecting the patterns in the query sequence is important to determine how to merge subsequent queries to achieve better response time by (1) obtaining at least the correct answer (can be more, but not less!), and (2) decreasing the performance implications of coding and executing the predicates, joins, or union in merged queries.

3 Query Merging Operators

Forming prefetch query sequences (partitions) requires merging the simple related queries to bring the data that is requested and that will be requested in the future at once. We study three different query merging operators:

Simple-Inner-Join (SIJ) contains columns for all of the projected tables involved in the queries. However, in any one row of the output, output columns from multiple queries will be filled. However, if there is a 1:N or M:N relationship among the join keys of merged queries, the result data size of the merged query will explode and data column values will be replicated many times. We further classify SIJ as:

- complete SIJ (c-SIJ), where both the outer query and the correlated inner query set is merged. e.g., the outer query Q_1 and the inner query set $\{Q_2, Q_3, Q_4\}$ is merged
- partitioned SIJ (p-SIJ), where only the queries in the same query set is merged, e.g., the queries in the set $\{Q_2, Q_3, Q_4\}$ is merged

Outer-Join (OJ) contains columns for all of the projected tables involved in the queries. However, in any one row of the output, output columns from multiple queries can be filled with NULLs. Tables or inline views in the FROM clause of an outer join can be classified as either preserved row or NULL-supplying. The preserved row refers to the table or inline view that preserves rows when there is no match in the join operation. Therefore, all rows from the preserved row table that qualify against the WHERE clause will be returned, regardless of whether there is a matched row in the join. The NULL-supplying table or inline view supplies NULLs when there is an unmatched row.

Outer-Union (OU) contains columns for all of the projected tables involved in the queries and an extra column as a query

Q_9		$Q_{10}(i_1)$	$Q_{11}(i_6)$	$Q_{12}(i_2)$	Q_{13}	
o_1	i_1	b_1	b_2	b_3	s_1	10
o_1	i_4	b_2	b_4	b_7	s_2	20
o_2	i_2					
o_3	i_3					

Table 1: Result table for Q_9, Q_{10}, Q_{11}, Q_{12} and Q_{13}, where *order_id* and *item_id* is returned for customer id 4000, *component_id* is returned for Q_{10}, Q_{11}, and Q_12 with input parameter values i_1, i_4, and i_2, respectively. Finally *supplier_id* and *qty_demanded* is returned for b_3.

number indicator. However, in any one row of the output, only output columns from a single query will be filled; all other columns will be null. Basically, the entire Outer Union query consists of the union over a set of smaller SQL queries.

Table 1 shows a sample output for the last five queries listed in Section 2.2. In the output table, we have three orders and the items ordered by a customer (Q_9). For three items ordered, we also bring the related components listed by (Q_{10}, Q_{11}, Q_{12}), then for one of the components, let's suppose for b_3, we have a supplier and the quantity listed by (Q_{13}). If we use SIJ to merge for Q_9 and Q_{10}, then we cannot bring the correct answer for the first query, where we either bring the components for all the items ordered or we only bring the data related to i_1. In the later case, we miss the other items ordered by orders o_1, o_2 and o_3, requested by the original query Q_9. Bringing more-than-requested data can be acceptable—depending on the size of this data, but less-than-requested is unacceptable. Under the perfect knowledge assumption, where we know the input parameters of the future queries, we can use left outer-join or outer-union to bring the exact data requested by the original queries. We further classify the data that is brought by the prefetch queries as (1) right data, which is the requested data; (2) false data, which will never be used; and (3) repeated data, which was already brought. If we decide to use SIJ to merge the queries, then we must also consider the size of the false and repeated data.

SIJ and left OJ can be used for merging queries with browse pattern, full OJ can be

SIJ (All items)			*SIJ (1 item)*		
o_1	i_1	b_1	o_1	i_1	b_1
o_1	i_1	b_2	o_1	i_1	b_2
o_1	i_4	b_2			
o_1	i_4	b_4			
o_2	i_2	b_3			
o_2	i_2	b_7			
o_3	i_3	b_5			

Table 2: Result table for the merged queries Q_9, Q_{10}, Q_{11}, and Q_{12}. The left three columns list the output that is formed via c-SIJ that brings all the items and their related components ordered by a customer; the right three columns list the same output for only one item ordered by a customer

Left OJ			*Outer-Union*			
o_1	i_1	b_1	9	o_1	i_1	null
o_1	i_1	b_2	9	o_1	i_4	null
o_1	i_4	null	9	o_2	i_2	null
o_2	i_2	null	9	o_3	i_3	null
o_3	i_3	null	10	null	null	b_1
			10	null	null	b_2

Table 3: Result table for the merged query Q_9 and Q_{10} via left OJ(Outer-Join) and OU(Outer-Union). For the OU, we also add an extra column to index the output with the query number

used for the queries with either input or output pattern, and OU can be used for all patterns. Table 3 lists the output of the prefetch query $\{\{Q_9, Q_{10}\}\}$ formed via left OJ, and OU.

Among the above query merging operators, full OJ is the most expensive one, because it includes SIJ, projection, difference, and union operators. We replace full OJ operator by the OU operator, because it is a more basic, less-costly operator and can do the same job. We explain the cost model for each operator in the Section 5. We can also use inline views while constructing queries with outer join. The inline view is a construct in SQL, where you can place a query in the SQL FROM, clause, just as if the query was a table name. A common use for inline views is to simplify complex queries by removing join operations and condensing several separate queries into a single query. Inline views are evaluated at runtime, and unlike normal views are not stored in the data dictionary; they're effectively named sub-queries that derive their rows at run-time during the execution of the outer query.

4 Detection of Parameter Correlations

To detect the query patterns among queries in a given query set, we first need to monitor the application data request stream, and track the input and output parameters of queries in the application's data request stream. This application data request stream can be either represented statically as a trace file, e.g., JDBC trace driven by the DB2 Universal JDBC driver DB2SystemMonitor interface, or dynamically via SQL-Relay [19, 20], which is a persistent database connection pooling, proxying and load balancing system between the client application and data server.

After tracking the input and output parameter correlations of the queries, we use a list of the following data structures to store the query features and the pattern correlations among queries:

- I_i^j holds the table_name, column_name, value, value_provider data for the j_{th} input parameter of the i_{th} query. The *value* variable holds the value of the input parameter, and *value_provider* holds the correlated output parameter variables represented with O_m^n, which means that n_{th} output column of the m_{th} query provides the input.
- O_i^j holds the table_name, column_name data for the j_{th} output parameter of the i_{th} query.
- $|O_i|$ stands for the number of output attributes of the i_{th} query.
- $|I_i|$ stands for the number of input parameters of the i_{th} query.

We only consider the primary and foreign keys of the tables as entries in the input and output parameter vectors, because we only consider the small-sized queries that brings the data according to the traversal of the keys in the tables.

4.1 Building the Query Pattern List

We use two different data structures to represent the patterns that are observed in a given query set: (1) a three-dimensional vector $BP_{i,j,index}$ to represent the browse pattern correlations of the given query set with corresponding I and O vectors, and (2) a HashMap from an input value to a query set that has an input pattern for this value, which is represented as $IHM(value) = \{query\_list\}$. Each $BP_{i,j,index}$ holds a node with the following variables, which hold the information about the ind_{th} browse pattern correlation of the j_{th} output parameter of the i_{th} query:

- *table_name* is the name of the table of the queries that have browse pattern.
- *column_name* is the name of the column of the queries that have browse pattern
- *query_list* is the list of the index of the queries that have browse pattern.

$IHM(value) = \{query\_list\}$ is the mapping of an input value to a query set, which is used to list the input patterns among queries. We assume that each key column of the tables have a different value domain to have a such a mapping from input values.

We build the two main data structures BP and IHM to represent browse and input pattern correlations, respectively. We design an algorithm [6], where for a given query sequence with size n, starting from the last query in the sequence until reaching to the first query, that fills up the Vector and HashMap for each possible patterns that can be observed among the columns of the queries.

The algoritm, which is explained in detail in [6], for detecting the pattern correlations, listed, takes $O(n^2)$ time if the number of output columns and input parameters is small compared to the number of queries. By this algorithm, we first detect the browse pattern among the output columns and input parameters of the queries, and then we look for the input pattern among the input parameters of the queries. There can be multiple browse pattern correlation for an output column of any query, where each correlation is represented by an index in BP_ij. We do not store column or table information for input pattern correlations, because as explained in the merging algorithm, the queries with such correlations are merged via Outer Union operator, which does not require this information. However, this algorithm can be easily extended to hold such information.

After the building phase of BP and IHM, we initialize a query set graph, which is a directed graph represented by a double adjacency list, where each node has two type of links to other adjacent nodes: (1) vertical link connects the queries with browse pattern correlations, and (2) horizontal link connects the queries with input pattern correlations. Each node represents the index of the query, and each query can have multiple vertical and horizontal links, where each link also has an identifier data to point to the related indexes or keys of BP and IHM, respectively.

We will have the following entries in BP and IHM for the sample query sequence listed in Section 2.2:

- $BP_{1,1,1}$= $(AccountTransaction, at\_a\_num, \{2, 3, 4\})$
- $BP_{1,1,2}$= $(AccountProducts, ap\_a\_num, \{5\})$
- $BP_{1,1,3}$= $(AccountRelations, a\_num\_from, \{6\})$
- $BP_{1,2,1}$=$(Agreement, ag\_id, \{7, 8\})$
- $IHM(4000)$=$\{1, 8\}$
- $BP_{9,2,1}$= $(Bom, b\_assembly\_id, \{10, 11, 12\})$
- $BP_{12,1,1}$= $(Supplier\_component, comp\_id, \{13\})$
- $IHM(Q_1.a\_num_3)$=$\{4, 5, 6\}$

4.2 Merging the Queries

After determining the query merging operator and the queries that are to be merged, we use the $merge(lhs, rhs, pattern\_indicator, op\_type)$ function listed in Appendix A to merge them, which takes four arguments. The *op_type* denotes type of the operator, which can be complete SIJ, partitioned SIJ, left OJ, or OU. The complete SIJ option is used to merge the query with the queries in its $query\_list$ of the corresponding BP entry, and the partitioned SIJ is used only to merge the queries in the $query\_list$.

The $merge$ function merges multiple queries if the SIJ operator is used; otherwise it merges only two queries at a time. The $pattern\_indicator$ is the indicator to determine whether there exists a browse pattern or input pattern correlation among the queries that is to be merged; 0 is used for input pattern and 1 is used for browse pattern. It also indicates whether to use BP data structure to obtain the right-hand-side query set. The lhs is the left-hand-side query, which can be the outer query if flag is 1 or just the first query to be merged if the flag is 0. The rhs is the right-hand-side query set, which is the list of $\langle column\_index, index\rangle$ pairs if flag is 1, otherwise it is just a single query index, which will be the second query to be merged. The $\langle column\_index, index\rangle$ pairs are used to determine the $query\_list$ of the $BP_{lhs,column\_index,index}$. In the following $merge$ algorithm, we have the variables to store the select, from, and where part of query written in SQL (We only list the variables that is used to store *select* part of the queries, the same indices also apply for the *from* and *where* part of the queries written in SQL).

- $select_i$ is the *select* part of the individual query q_i
- $select_{i,j}$ is the *select* part of the merged query $q_{i,j}$
- $select_Q$ is the *select* part of the *query_list* of $BP_{i,column\_index,index}.query\_list$. Q also includes q_i if complete SIJ is used
- $inlineView_i$ is the inline view that corresponds to the individual query q_i, which is used to build the *from* part of the merged query. Each inlineView also has *select*, *from*, and *where* subparts, which are all initialized as $\emptyset$
- $name_i$ is the alias name of the inlineView of the query q_i
- $|O_{i,j}|$ is the number of the output columns of the merged query
- *query_identifier* is the index of the each individual query added to the *select* part of the merged query, which is used to distinguish the output tuples of each individual query
- ou_select_i is the select part of the each individual query, which is padded with query identifier and the other columns, to make it union with the other queries
- q_{merged} is the output of the algorithm, which is the merged query

Using the *merge* algorithm, we produce the following sample prefetch queries by merging the queries listed in Section 4.1:

- $\{Q_9, Q_10, Q_{11}, Q_{12}\}$ with complete SIJ:
  select
  $ol\_order\_id, ol\_item\_id, b\_component\_id$
  from Orders, Orderline, Bom
  where $o\_customer\_id$ = 4000 AND $o\_id = ol\_o\_id$ AND $b\_assembly\_id = ol\_item\_id$;
- $\{Q_10, Q_{11}, Q_{12}\}$ with partitioned SIJ:
  select $b\_component\_id$
  from Orders, Orderline, Bom
  where $o\_customer\_id$ = 4000 AND $o\_id = ol\_o\_id$ AND $b\_assembly\_id = ol\_item\_id$;
- $\{Q_{12}, Q_{13}\}$ with left OJ:
  select $b\_component\_id, qty\_demanded$
  from
  (select $b\_component_id$ from Bom where $b\_assembly\_id$= $Q_8.ol\_item\_id_3$) as m_Bom
  left outer join
  $Supplier\_component$ on $m\_Bom.b\_component\_id$ = $comp\_id$ AND $comp\_id$= $Q_{11}.b\_component\_id_1$;
- $\{Q_{12}, Q_{13}\}$ with OU:
  select 12, $b\_component\_id, NULL, NULL$
  from Bom where $b\_assembly\_id$= $Q_8.ol\_item\_id_3$
  union all
  select
  13, $NULL, supplier\_id, qty\_demanded$
  from $Supplier\_component$ where $comp\_id$= $Q_{11}.b\_component\_id_1$

5 Cost Model for Merging Operators

The query pattern will classify the query and under the appropriate classification, the cost of using a specific query merging operator determines the efficiency of merging. We evaluate the execution cost and fetching all the rows of a SQL query q as follows as in [7]:
$cost(q)$=U_0 + $server\_cost(q)$ + $transfer\_cost(|R|, \|R\|)$

For an individual query, $server\_cost(q)$ gives the server cost of a query in terms of physical and logical disk page accesses as estimated by RDBMS. The overhead associated with a single *open* request is also measured and represented by U_0. The $transfer\_cost(|R|, \|R\|)$ estimates the transferring of $|R|$ rows of $\|R\|$ bytes, which is the average row length for q. The transfer cost estimates bringing the data from data server process space to client application process space, which can also include the network latency of TCP/IP protocol if the client and the server resides on different computers, but does not include the latency of printing the data.

For each prefetch query q_{merged}, depending on the appropriate merge operator used, we re-estimate the $server\_cost(q_{merged}) + transfer\_cost(|R|, \|R\|)$. To explain the details for estimating the cost of executing prefetch queries, which are merged with one of the three merging operators, we first list the following parameters and data structures:

- Local query parameters:
  - I is explained in Section 4
  - O is explained in Section 4
  - $|R_i|$ stands for the number of output tuples of the ith query
  - $\|R_i\|$ stands for average output tuple size of the ith query
  - $|O_i|$ is explained in Section 4
  - $|I_i|$ is explained in Section 4
- Local database parameters:
  - $J(a, b)$ stands for the average join ratios among keys of different tables, where $J(a, b)$ represents the join rate between key a and key b
  - $T(table\_name)$ stands for the table sizes of the given database in terms of number of tuples
  - $B(table\_name)$ stands for the table sizes of the given database in terms of number of data pages
- Local query set parameters:
  - n is the number of the queries in the given query sequence.
  - BP is the three dimensional vector of browse pattern relationships in a query set, which is explained in Section 4
  - IHM is an HashMap for the input pattern relationships in a query set, which is explained in Section 4
  - $Nbp_{i,j,index}$ is the number of queries in the query list of $BP_{i,j,index}$
  - $bf_{i,j,index}$ is the branch factor, which is the ratio of $Nbp_{i,j,index}$ to the number of different output tuples, which is estimated as $J(I_i^1, O_i^j)$. Then we have the following formula: $bf_{i,j,index} = \frac{Nbp_{i,j,index}}{J(I_i^1, O_i^j)}$
- Global database parameters:
  - L is the network latency in terms of bits per second
  - Buf is the size of the buffer cache in terms of data pages

	c_id	a_num	ag_id
c_id	1	10	–
a_num	2	1	1
ag_id	–	1.5	1

Table 4: Average join ratios for some of the keys, which were used in Section 2.2

J table is one of the most important parameters in estimating the cost of executing merged queries. For example, in Table 4 $J(ag\_id, a\_num) = 1.5$, which means that there are 1.5 account number in average for each agreement number, and $J(a\_num, ag\_id) = 1$, which specifies that there is 1 agreement number in average for each account number. By producing J table, we can also derive indirect join ratios such as $J(c\_id, ag\_id) = 10$ according to Table 4. J can be also used to estimate the number of output tuples of a query, that does not have selective predicates (filter ratios), i.e., the predicate $account\_balance \leq \$1000.0$ while requesting the accounts of a customer. For the Q_1 listed in Section 2.2, we have $bf_{1,1,1} = \frac{3}{10} = 0.3$, and $bf_{1,2,1} = \frac{2}{10} = 0.2$.

5.1 Cost Formula for SIJ

If there is an entry as $BP_{i,column\_index,index} = (table\_name, column\_name, \{j, \ldots\})$, where j is the member if the $query\_list$, then the number of output tuples for the merged query ($|R_{i,j}|$) is $|R_i| \times |R_j|$, and the $\|R_{i,j}\|$ is $|R_i| + |R_j|$. In estimating the server cost of the prefetch query, we consider the following situations:

- The number and size of the shared tables,
- The size of the tables $O_i^{column\_index}.table\_name$ (let's denote it as t_i) of q_i and $table\_name$ of q_j (let's denote it as t_j), which are the tables that are used to build the extra join predicate of the prefetch query.

We assume that underlying query optimizer is responsible for the join optimization of the prefetch query. We also assume that tables t_i and t_j are joined via nested-loop join method. Nested-loop join is the efficient method to merge if the size of the outer query is not big and one of the queries has an index on the join key. So based on the general block nested loop join cost formula, which is $(N \times (1 + \frac{M}{Buf-2}))$, if t_i and t_j has N and M pages, respectively with the buffer size of Buf, and there exits no indexes on the join attributes. Depending on the size of the buffer the join cost can be reduced to $N + M$ page accesses, or depending on the existence of index on the inner table (t_j), the cost is reduced to $N + (|R_{t_i}|) \times c$, where $|R_{t_i}|$ is the number of tuples in table t_i and c is the cost of looking up a tuple in t_j using the index.

In terms of the initial overhead U_0, instead of having $1 + Nbp\ U_0$, we just have one U_0 in q_{merged}.

5.2 Cost Formula for left OJ

If the left OJ is used to merge the queries, then we use J to derive $|R_{i,j}|$ and $\|R_{i,j\|}$. For the entry $BP_{i,column\_index,index} = (table\_name, column\_name, \{j\})$, we use $J(I_i^1.column\_name, O_i^{column\_index}.column\_name)$ to estimate number of different output $column\_name$ values and use $\jmath$ to estimate the average number of tuples with the specific $column\_name$ value, where $\jmath$ is defined as: $\lceil \frac{|R_i|}{J(I_i^1.column\_name, O_i^{column_index}.column\_name)} \rceil$

We also assume that 1 byte null indicator is used for null column values of the prefetch query. $|R_{i,j}|$ is then estimated as $\jmath \times (J(a_j, O_j^{|O_j|}) - 1) + |R_i|$, and $\|R_{i,j}\|$ is estimated as $(\frac{\jmath \times |R_j| \times (\|R_i\| + \|R_j\|) + (|R_i| - \jmath) \times (\|R_i\| + |O_j| \times bytes(null))}{|R_{i,j}|})$, where $bytes(null)$ function estimates the size of the null field indicator which is one byte at most of the commercial data servers that use some compression techniques, and a_j is $BP_{i,column\_index,index}.column\_name$.

Estimating the server cost of a prefetch query requires considering the similar situations as listed for SIJ. In addition to these cases, we also consider the sizes of the inline views in addition to the sizes of the tables while joining, which are estimated by $|R|$ parameter of the queries.

5.3 Cost Formula for OU

Estimating the cost formula for OU is the easiest one, where $|R_{i,j}|$ is simply $|R_i| + |R_j|$, and $\|R_{i,j}\|$ is estimated as $\frac{|R_i| \times (\|R_i\| + |O_j| \times bytes(null)) + |R_j| \times (\|R_j\| + |O_i| \times bytes(null))}{|R_i| + |R_j|}$. We only consider the transfer cost of the result of query, while merging the queries via OU.

6 How to Find Efficient Prefetch Queries

Our approach to reduce the response time of the applications depends on the idea of merging the queries as an alternative of prefetching. During the process of merging queries, we use the query optimizer in the RDBMS that stores persistent objects, and we do not consider the problem of answering the original query from the merged queries (for possible solutions to this problem see, e.g., [8]). The three main tasks, our system has to aware of, are:

- What are the generic domain-independent guidelines for merging and which parameters are useful to determine these guidelines?
- What is the complexity of merging, which includes determining the queries to be merged, initializing and re-estimating the parameters of the queries and query set each time after merging some of the queries in the query set, and adding new merged queries to the query set?

- What is the distance between the benefit of using original query sequence and estimated benefit of merging?

In addition to the main task, deriving generic domain-independent guidelines requires the following tasks:

- generating meaningful combinations of query sequences with any of the patterns for the given database
- testing the queries to understand when and how they become a candidate mergable query with the other candidate queries
- evaluation of the parameter values

We use the extended SPECJ benchmark data model to generate meaningful combinations of query sequences, which is explained in the next section. We also mention the process of how the benefit of merging is estimated by the guidelines after the evaluation of parameter values.

6.1 Extended SPECJ Benchmark

We generate use cases and construct a testbed to experiment with the parameters that affect the cost of prefetch queries. We extend and adapt SPECJ benchmark data model to accommodate variety of use cases that is required to answer *how much*, *how deep*, *in what direction* to read ahead. SPECJ is a popular benchmark that is commonly used for testing the performance of commercial Java enterprise application servers. We extend and adapt the data model of this benchmark in the following way:

- add parameters to iteratively change multiplicity of relationships;
- include all different type of relationship multiplicities, e.g., 0..1 to 1, 1..* to 1;
- adapt entity structure to accommodate both breadth-first and depth-first traversals;
- include keys with different number of attributes and foreign keys w/o index;

Figure 1 shows the main entities of our data model.

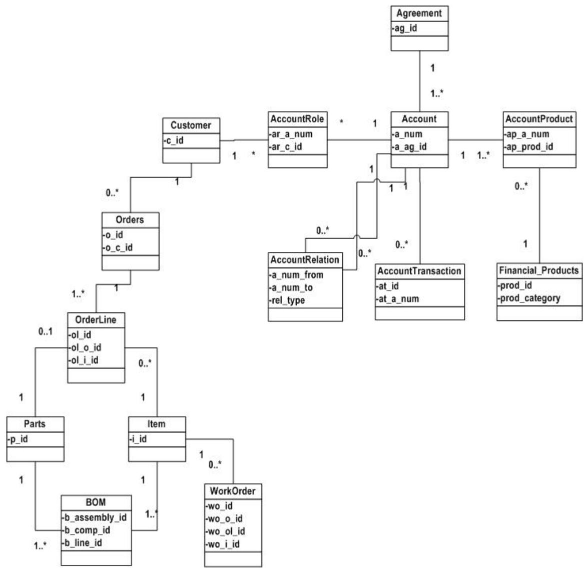

Figure 1: Main entities in extended SPECJ benchmark data model with their keys

6.2 Efficiency of a Query

We define the real efficiency of merged query $q_{i,j}$ as $e_{i,j}$, which is calculated as:
$e_{i,j} = \frac{Cost(q_i)+Cost(q_j)}{Cost(q_{i,j})}$

Basically $e_{i,j}$ is the ratio of the cost of processing the queries individually to the cost of processing the queries together. If $e_{i,j} > 1$, then processing the queries together is cheaper. We also define $e_{i,j} = 0$ if two queries are disjoint, which means that there is not any pattern correlation. Real efficiency is computed after the execution of the queries.

We also define the estimated efficiency of merged query $f_{i,j}$, which is the multiplication of all the guideline factors, which are mentioned in Section 7. The six proposed guideline factors are denoted as gf_i (i ranging from 1 to 6). The estimated efficiency is given by $f_{i,j} = \prod_{i=1}^{6} gf_i$. Each guideline factor has a basic lookup table to determine its value, so that the efficiency of the merging is estimated. We set the unrelated guideline factors for each merging operator to 1.0, e.g., if SIJ is used to merge the queries, then $gf_6 = 1.0$, or if OU is used to merge, then $gf_1 = gf_2 = gf_3 = gf_4 = 1.0$.

In principle, additional guideline factors can be specified based on additional experiments. Doing so may increase the accuracy of the estimated efficiency with respect to the actual efficiency.

6.3 Basics of the Algorithm to Find Prefetch Queries

In each step of the algorithm, we can merge one query with multiple query sets via SIJ, or we can merge one query with another query via left OJ, or OU. So even there exist three queries that have the same input pattern correlation, we merge them one by one returning to the reestimation of the parameters phase (step 3) in the following algorithm. We apply the following steps to find the sequences of prefetch queries:

1. **Step 1:** By tracking the variable correlations in the application, the pattern correlations are detected
2. **Step 2:** The query set graphs including pattern correlation information is initialized
3. **Step 3:** By using the database parameters, and detected pattern information, local query and query set parameters are determined or recalculated
4. **Step 4:** For each connected graph in the set of query set graphs, in a bottom-up manner (from the leaf level to root level), estimate the efficiencies of merging queries (f_{merged}) with the appropriate operators
   (a) **Step 4.1:** If $Nbp > 1$, then consider SIJ operator for merging
   (b) **Step 4.2:** Determine complete or partitioned SIJ for merging, and the query subset(s) that are to be merged
   (c) **Step 4.3:** If the query q_i has $Nbp = 1$ and $|R_i| > 10$, then consider left OJ or OU operator for merging with q_j
   (d) **Step 4.4:** If input pattern exists, consider OU operator for merging q_i and q_j
   (e) **Step 4.5:** Replace the queries that are merged with the new merged query, and go to **Step 3**

7 Experimental Results and Guideline Factors

Our main task during designing the experiments was to understand the effect of parameters to query elapsed time via exploring meaningful query sequences, which were derived from the extended and adapted SPECJ benchmark data model. In the experiments, we used a 1.7 GHz Pentium M, 512 MB memory, 60 GB 5400 rpm hard disk with Windows XP operating system, and ran tests against IBM DB2 v8 Enterprise Data Server by using the interactive SQL engine (DB2 Control Center) to execute the queries. For the guidelines, we did not consider the network latency and JDBC overhead. We reset the content of the buffer cache before each query execution by submitting unrelated "dummy" queries to fill the buffer. Initially we populated data to the SPECJ data model with an orders injection rate of 150, and default buffer cache size was 256 4kB pages.

We determined six guideline factors used to predict the efficiency of merging. Although, input pattern correlation is common in all query sets, it is the browse pattern correlation that provides more benefit when used with SIJ operator, which is due to the elimination of more database requests.

Any two queries can be merged with OU operator. However, the efficiency of using OU operator is limited as explained by gf_6.

7.1 GF 1: When to Merge under Browse Based on Number of Queries

Guideline factor 1, abbreviated gf_1, is used to examine the effect of $|R|$, Nbp, and bf parameter values to the efficiency of query sets merged via SIJ. In Table 5, we consider merging multiple queries with SIJ (only considering multiple queries reveals the efficiency of using SIJ operator) and examine the Nbp values that make $gf_1 = 1.0$, which means that for the given $BP_{i,column\_index,index} = (table\_name, column\_name, \{query\_list\})$ entry, what is the required size of the $query\_list$ ($Nbp_{i,column\_index,index}$) for different $|R_i|$ and

$|R_j|$ values where gf_1 is 1.0 and j is the index of one of the queries in the $\{query\_list\}$. As Table 5 lists, if both queries have $R \geq 100$, then the query set should not be merged via SIJ even when bf is 1.0; because the required Nbp value is 112.79 (the redundant data explodes due to result of the join), which is impossible if $|R_i|$ returns 100 tuples. However, merging is always the efficient way if $|R_i|$ is 1.0. Also if both queries return 10 tuples, then required Nbp value is 2.57, so we need $bf = \lceil \frac{2.57}{10} \rceil = 0.3$ to make the processing of queries together efficient.

$\lvert R_i\rvert$	$\lvert R_j\rvert$	Nbp	gf_1
1	1	0.68	1.0
1	10	0.70	1.0
1	100	0.78	1.0
10	1	1.16	1.0
10	10	2.57	1.0
10	100	5.01	1.0
100	1	7.27	1.0
100	10	15.80	1.0
100	100	112.79	1.0

Table 5: The value of the Nbp parameter that makes the guideline factor 1.0. Assumptions are $\|R_i\| = 8$ bytes, $\|R_j\| = 8$ bytes, there exists index on both join tables, whose sizes are 10000 tuples

In Table 6, we examine the effect of branch factor for different Nbp values. In the following table, we just show the index of the query as Nbp_i, and eliminate the *column_index* and *index* for the sake of clarity. As the table lists, if the Nbp_i value increases with the same bf_i value, e.g., the gf_1 is 1.27, when the *query_list* have 20 queries, and it is lower than 1, when the *query_list* has just 2 queries. Below table uses the fixed values such as $|R_j| = 10$ (we already gave the efficiency estimates for different $|R|$ values in Table 5), $\|R_i\| = 8$ bytes, and $\|R_j\| = 8$.

7.2 GF 2: Complete or Partitioned SIJ

Guideline factor 2 or gf_2 is used to determine whether to use complete SIJ (c-SIJ) or partitioned SIJ (p-SIJ) to merge the query set. It

$\frac{Nbp_i}{\lvert R_i\rvert} = bf_i$	gf_1
$\frac{2}{10} = 0.2$	0.94
$\frac{20}{100} = 0.2$	1.27
$\frac{3}{10} = 0.3$	1.12
$\frac{30}{100} = 0.3$	1.96
$\frac{4}{10} = 0.4$	1.53
$\frac{40}{100} = 0.4$	3.19

Table 6: The value of the guideline factor with different $\frac{Nbp_i}{|R|_i} = bf_i$ values. The i variable is the query index of the Nbp and bf vectors

can be observed from Table 7 that, for small ($\|R_i\| \times |R_j|$) values, gf_2 for complete SIJ is higher, where i is the query index that has the corresponding BP entry and j is the query index of one of the queries in the *query_list* of the same entry. The bottom line in using the complete SIJ is when we have (270 bytes $\times$ 1) or (24 bytes $\times$ 10), and $bf = 0.3$, $\|R_j\| = 8$ bytes, and $|R_i| = 10$. The partitioned approach can be used, when bringing the data requested by q_i eliminates the efficiency of merging with complete SIJ, which is the case when $24 \leq \|R_i\| < 1KB$ according to Table 7.

$\|R_i\|(bytes) \times \lvert R_j\rvert$	gf_2(c-SIJ)	gf_2(p-SIJ)
8×1	1.77	1.26
40×1	1.34	1.21
270×1	1.00	1.18
$4K \times 1$	0.09	1.14
8×10	1.12	1.07
24×10	1.03	1.06
40×10	0.97	1.05
$1K \times 10$	0.14	0.99
$4K \times 10$	0.02	0.97
8×100	0.73	0.67

Table 7: The guideline factor for complete SIJ and partitioned SIJ approach for different $\|R_i\|(bytes) \times |R_j|$ values. The assumptions are $bf = 0.3$, $\|R_j\| = 8$ bytes, and $|R_i| = 10$

7.3 GF 3: Effect of Table Sizes and Indexes to Merging

Guideline factor 3 or gf_3 is used to determine the effect of table sizes and index existence on the join keys of the extra join predicate, when

SIJ or left OJ is used to merge the q_i and the corresponding query set assuming buffer size is 256 $4K$ pages. In Table 8, the quadruple variable i, t_i, j, t_j denotes the indexes of the queries with i and j, and the size of the tables that are joined as a side effect of merging with t_i and t_j. For example, for the merged query $Q_{8,9,10,11}$ formed by merging the individual queries in Section 4.1 via SIJ, we have the new join predicate $Bom.b\_assembly\_id = Orderline.ol\_item\_id$, where $t_i = Orderline$ and $t_j = Bom$. As listed by the table, the increase in the size of the join table slightly decreases the gf_3. If no index exists on any of the join keys, then queries should not be merged with SIJ or left OJ. The gf_3 is largely depends on the query optimizer's decisions such as using index-based nested loop joins.

i, t_i, j, t_j	gf_3
$(1, 10000, 1, 10000)$	1.12
$(1, 10000, 1, 100000)$	1.06
$(1, 100000, 1, 10000)$	1.10
$(1, 100000, 1, 100000)$	1.06
$(1, 100000, 1, 1000000)$	0.96
$(1, 10000, 0, 10000)$	0.83
$(0, 10000, 1, 10000)$	0.77
$(0, 10000, 0, 10000)$	0.39

Table 8: The guideline factors for quadruple variable i, t_i, j, t_j, where i is the index flag for outer join table, j is the index flag for inner join table (1 means index exist, 0 means index does not exist), t_i, and t_j is the size of the outer and inner join tables, respectively, in terms of number of tuples assuming nested loop join is used to join the two tables. The assumptions are $bf = 0.3$, $|R_j| = 10$, $|R_i| = 10$, $\|R_i\| = 8$ bytes, $\|R_j\| = 8$ bytes, and buffer size is 256 $4KB$ pages

7.4 GF 4: Merging with More than One Query Sets

Guideline factor 4 or gf_4 is used to determine the effect of merging more than one query set with the same query (multiple BP entries of the same query q_i). In Table 9, $bf_{i,;,1}$ denotes the branch factor of the $BP_{i,column\_index,1}$, where column_index is not specified. We also fix the values as $|R| = 10$, and $\|R\| = 8$ bytes for all queries. If both $bf \geq 0.3$, then the efficiency of merging two query set is higher than efficiency of merging one inner query set. The bottom line in merging two query sets is the bf pair (0.3, 0.2), whose gf_4 value is lower than the pair (0.4, 0.1).

The gf_4 is the guideline factor that is most amenable to extension such as analyzing the effect of merging query sets from BP entries with different *column_index* values and from different *index* values, which effects the number of extra join predicates and the variety of tables that are joined.

$bf_{i,;,1}$	$bf_{i,;,2}$	gf_4^1	gf_4^2	$gf_4^{\{1,2\}}$
0.5	0.5	1.76	1.76	2.09
0.3	0.3	1.13	1.12	1.26
0.3	0.2	1.13	0.94	1.02
0.4	0.1	1.50	0.67	1.08
0.3	0.1	1.13	0.67	0.88

Table 9: The value of the guideline factors, when two query sets with of different bf entries with index 1 and 2 are merged with a query, assuming $|R| = 10$ for both query sets and the query q_i. The $gf_4^{\{1,2\}}$ denotes the guideline factor for the query that is formed via merging the two query sets

7.5 GF 5: Using Left OJ or OU

Guideline factor 5 or gf_5 is used to determine whether to left OJ or OU is the effective operator for merging in cases where SIJ is not appropriate, which are the cases when the multiplication of all previous guideline factors are smaller than 1.0 and $BP_{i,column\_index,index}$ entry has only one query q_j in its *query_list*. As listed by Table 10, Left OJ is better option than OU if $|R_i|$ is in the order of 10 tuples and $\|O_j\|$ does not have more than 16 attributes, where $\|O_j\|$ affects the size of the NULL data. The high $|R_i|$ values increase the server cost of the left OJ due to the join of inline views, so decrease the gf_5 for left OJ. OU can be used, if $|R_i|$ is in the order of 100 tuples and $\|O_j\|$ does not have more than 8 attributes.

$\|O_j\| \times \|R_i\|$	gf_5(left OJ)	gf_5(OU)
1×10	1.19	1.16
8×10	1.16	1.11
64×10	0.75	0.66
1×100	1.11	1.15
8×100	0.91	1.03
64×100	0.36	0.58
1×1000	0.87	1.01
8×1000	0.44	0.89
64×1000	0.18	0.51

Table 10: The guideline factors for left OJ and OU approach for different $\|O_j\| \times |R_i|$ values where Nbp value for the corresponding BP entry of q_i is 1 and $\|O_j\|$ is the number of output attributes of q_j of the *query_list*, assuming each output attribute requires 4 bytes without NULL and 1 byte with NULL (null indicator), $|R_j| = 10$, $\|R_i\| = 8$ bytes

7.6 GF 6: Efficiency of OU

Guideline factor 6 or gf_6 is used to determine whether to use OU as the effective operator for merging when input pattern or browse pattern between two queries q_i and q_j are observed. The bottom line in using OU is, when $(\|O_i\| \times |R_j| + \|O_j\| \times |R_i|)$ has the values $(16 \times 10 + 8 \times 100 = 960)$, assuming all columns require 4 bytes and NULL column value requires 1 byte.

$\|O_i\| \times \|R_j\| + \|O_j\| \times \|R_i\|$	gf_6(OU)
$1 \times 10 + 1 \times 10$	1.21
$1 \times 100 + 1 \times 100$	1.09
$8 \times 10 + 8 \times 10$	1.11
$8 \times 100 + 8 \times 100$	0.95
$8 \times 10 + 8 \times 100$	1.03
$16 \times 10 + 8 \times 100$	1.01
$8 \times 100 + 1 \times 10$	1.04

Table 11: The guideline factor for OU approach for different $\|O_i\| \times |R_j| + \|O_j\| \times |R_i|$ values where $\|O_i\|$, and $\|O_j\|$ is the number of output attributes of two queries q_i and q_j with input pattern or browse pattern, assuming each output attribute requires 4 bytes without NULL and 1 byte with NULL (null indicator). $|R|$ stands for the number of output tuples for each query

8 Sample Query Sets and Using Guidelines

We use query templates to represent the tested set of queries, where each query template denotes the queries, which have the same *from* part and has the same *where* part with the same input parameters but with different input parameter values. We generate query sequences that consist of minimum 5 queries (we used no maximum value because of the parameter Nbp) by using the following query templates, which was derived from the extended SPECJ data model:

T_1 : select *a_num*, *ag_id* from AccountRole, Account where *ar_customer_id* = c_id_x and *ar_a_num* = *a_num*

T_2 : select *at_id*, *at_amount* from Account-Transaction where *at_a_num*= a_num_x

T_3 : select *ap_product_id* from AccountProducts where *ap_a_num*= a_num_x

T_4 : select *a_num_to* from AccountRelations where *a_num_from*= "a_num_x"

T_5 : select *ap_product_id*, *pc_type_id* from AccountProducts, ProductCategory where *ap_a_num*= a_num_x and *ap_p_id* = *pc_p_id*

T_6 : select *a_num* from Account where *a_ag_id*= ag_id_x

T_7 : select *ar_customer_id* from AccountRole where *ar_a_num*= a_num_x

T_8 : select *agreement_amount* from Agreement where *ag_id*= ag_id_x

T_9 : select *asset_id* from AgreementAssets where *ag_id*= ag_id_x

T_{10} : select *asr_asset_id*, *agrass_ag_id* from Asset-
Role, AgreementAssets where *asr_c_id*= c_id_x and *asr_asset_id* = *agrass_asset_id*

T_{11} : select * from Products where *p_id*= $product\_id_x$

T_{12} : select *asr_c_id* from AssetRole where *asr_asset_id*= $asset\_id_x$

T_{13} : select * from Assets where $asset\_id$= $asset\_id_x$

In the query template sequences, we have a customer, who can have many accounts, which is stored in *AccountRole*. Also each account can be shared by many customers. Each account can have many account transactions, can be related with other accounts (stored in *AccountRelations*), can have many attached products that define the properties of the account (stored in *AccountProduct*), and can be used according to only one agreement. Each agreement is accepted, if the specific assets are provided in *AgreementAssets* by the customers. Each asset can be owned by many customers and each customer can have many assets (stored in *AssetRole*). Also products can be member of many product categories.

The query templates include input parameter values in the format of a_num_x, ag_id_x and so on, where x can be any values denoting the output parameter value of the dependent query. We denote the queries that use the same query template but with different O parameter as $T_i[O^n_m]$, where i denotes the query template index, n denotes the original query index in the query sequence, and m denotes the output column index as explained in Section 4. We also use BP and IHM structures for query templates to represent the pattern correlations. As a shortcut, we use $T_i[*]$ to denote a extended template that brings all the columns in the answer. In the following query sequences, all run times are shown in terms of 10^{-4} seconds. The value 6.9 means that the runtime is 6.9×10^{-4} seconds.

The sequence $\{T_1^{IHM(cid\_4000)}$, $T_2^{BP_{1,1,1}}[at\_id]$, $T_9^{BP_{1,2,1}}\}$ uses three query templates, but consists of 5 queries, which actually represents $\{\{Q_1\},\{Q_2^1\},\{Q_2^2\},\{Q_2^3\},\{Q_9^1\}\}$, because $Nbp_{1,1,1} = 3$, and $Nbp_{1,2,1} = 1$. For this sequence, the other parameters are $|R_{T_1}| = 10$, $|R_{T_2}| = 10$, $|R_{T_9}| = 10$, $\|R_{T_1}\| = 8$ bytes, $\|R_{T_2}\| = 4$ bytes, $\|R_{T_9}\| = 4$ bytes, all tables contain 10.000 tuples, and buffer size is 256 $4KB$ pages. The default execution time (without merging) of this sequence is 29.1. By using our guidelines, we determine to merge T_1 with T_2 via complete SIJ, so that the execution time of the prefetched query sequence $\{T_1, T_2\}, \{T_9\}$ is 26.5. We merge T_1 with T_2, because $bf_{1,1,1} = 0.3$, $Nbp_{1,1,1} = 3$ (gf_1), there exists indexes, table sizes of the new join condition in the merged query are small (gf_3), and we use complete SIJ, due to the small size of redundant data that is returned (gf_2). We do not merge both query sets T_2 and T_9 with T_1 via complete SIJ (gf_4), because bf pair is (0.3,0.1), which provides poor efficiency. Finally, we do merge the merged query $\{T_1, T_2\}$ with T_9 via OU by using gf_5 and gf_6, where $\|O_{T_9}\| = 1$, and $|R_{T_{1,2}}| = 100$. The total execution time of the result prefetched query is 24.7, which is $\{T_1, T_2, T_9\}$.

By using the above query templates, but with different $Nbp_{1,2,1}$, which is 3, with the table sizes *AccountRole* = 100000, *Account* = 10000, *AccountTransaction* = 100000, *AgreementAssets* = 100000 and having indexes among join keys, the best execution strategy becomes merging both query sets T_2 and T_9 with T_1 via complete SIJ, where the execution time is reduced from 44.3 (without any merging) to 39.9. If *AccountTransaction* has 1000000 tuples, then the best execution alternative still remains same according to gf_3 and gf_4, but with less efficiency: the execution time only reduces from 46.4 (without any merging) to 43.3. However, if we have *AccountTransaction* has 1000000 tuples, and *Account* has 100000 tuples, then the best execution strategy is $\{T_1, T_9\}, \{T_2\}$, where query sets T_1 and T_9 is merged via complete SIJ and the three queries in $\{T_2\}$ is submitted individually due to high server cost of processing the extra join condition.

We test another query template sequence $\{T_1^{IHM(cid\_4000)}$,$T_5^{BP_{1,1,1}}$,$T_{11}^{BP_{5,1,1}}\}$, which corresponds to the query sequence $\{\{Q_1\},\{Q_5^1\},\{Q_{11}^1\},\{Q_{11}^2\},\{Q_5^2\},\{Q_{11}^3\},$ $\{Q_{11}^4\},\{Q_5^3\},\{Q_{11}^5\},\{Q_{11}^6\}\}$, with the parameter values $bf_{1,1,1} = 0.3$, $bf_{5,1,1} = 0.4$, $\|R_{T_{11}}\| =$ 40 bytes, $|R_{T_1}| = 10$, $|R_{T_5}| = 10$, $|R_{T_11}| = 1$. The default execution time is 56.0 for this sequence. Then by using the values bf, and the answer size of the inner queries (gf_1 and gf_2), we merge the queries $\{Q_5^1, Q_{11}^1, Q_{11}^2\}$ in each sub-sequence via complete SIJ. However, we do not merge the three prefetched queries with Q_1, due to size of the inner prefetched queries, which is $(40 + 8) \times 10$ (gf_2). The execution time of the prefetched query sequence $\{\{Q_1\}, \{Q_5^1, Q_{11}^1, Q_{11}^2\}, \{Q_5^2, Q_{11}^3, Q_{11}^4\}, \{Q_5^3, Q_{11}^5, Q_{11}^6\}\}$, which consists of 4 queries now, is 41.9. For the same sequence if $bf_{5,1,1} = 0.2$, then the best execution alternative is $\{\{Q_1, Q_5^1, Q_5^2, Q_5^3\}, \{Q_{11}^1\}, \{Q_{11}^2\}, \{Q_{11}^3\}, \{Q_{11}^4\}, \{Q_{11}^5\}, \{Q_{11}^6\}\}$, with execution time of 52.7, where prefetched query is formed via complete SIJ (gf_1 and gf_2). Here the prefetched query $\{\{Q_1, Q_5^1, Q_5^2, Q_5^3\}$ provides better efficiency than $\{Q_5^1, Q_{11}^1, Q_{11}^2\}$ according to gf_1, and partitioned SIJ is not efficient for forming $\{Q_{11}^1, Q_{11}^2\}$ in each sub-sequence because $|R_{T_{11}|} = 1$.

The default execution time of the query template sequence $\{T_9^{IHM(agid\_000001)}[*], T_{12}^{BP_{9,1,1}}, T_{13}^{BP_{9,1,2}}\}$, where $\|R_{T_9}\| = 40$, $\|R_{T_{12}}\| = 4$, $\|R_{T_{13}}\| = 40$ bytes, $|R_{T_9}| = 10$, $|R_{T_{12}}| = 10$, $|R_{T_{13}}| = 1$, $\|O_{T_9}\| = 5$, and both bf values are set to 0.3, is 57.7. The best execution strategy for this sequence is formed, when T_9 and the query set T_{13} is merged via complete SIJ and the three queries in $\{T_{12}\}$ is merged via partitioned SIJ, which gives the execution time 48.9 with two prefetched queries. In this scenario, we do not merge all the queries in the original query sequence as one query via complete SIJ, because p-SIJ is more efficient way of merging $\{T_{12}\}$ after T_9 and the query set T_{13} is merged (gf_2). However, if we increase the $\|R_{T_9}\|$ to 270 bytes, then best execution strategy is formed by merging T_9 with only one query Q_{12}^1 in T_{12} via left OJ (gf_5), submitting the other two queries in T_{12} individually, and merging the three queries in $\{T_13\}$ via p-SIJ, which gives the total execution time 52.5 instead of 59.1.

Another query template sequence $\{T_1^{IHM(cid\_4000)}, T_2^{BP_{1,1,1}}[at\_id], T_{10}^{IHM(cid\_4000)}, T_6^{BP_{10,2,1}}, T_2^{BP_{6,1,1}}\}$ with $bf_{1,1,1} = 0.3$, $bf_{10,2,1} = 0.1$, $bf_{6,1,1} = 0.1$, and $|R| = 10$ for all queries, has the default execution time 43.6. By using gf_1, we merge T_1 and the query set $T_2^{BP_{1,1,1}}$, then by using gf_5, we first merge T_6 and $T_2^{BP_{6,1,1}}$ via left OJ, then we merge T_{10} and $\{T_6, T_2\}$ again via left OJ to form $\{T_{10}, T_6, T_2\}$; and finally this last prefetched query can be merged with $\{T_1, T_2\}$ via OU (gf_6), which reduces the total execution time to 34.1. As another query template sequence, we consider $\{T_1^{IHM(cid\_4000)}, T_2^{BP_{1,1,1}, IHM(a\_num\_1)}, T_3^{BP_{1,1,2}, IHM(a\_num\_1)}, T_{11}^{BP_{3,1,1}}, T_4^{BP_{1,1,3}, IHM(a\_num\_1)}, T_{10}^{IHM(cid\_4000)}, T_6^{BP_{10,2,1}}\}$ with $bf_{1,1,1} = 0.1$, $bf_{1,1,2} = 0.1$, $bf_{1,1,3} = 0.1$, $bf_{3,1,1} = 0.3$, $bf_{10,2,1} = 0.3$, and $|R| = 10$ for all queries, which has the default execution time 68.1. By using gf_1 and gf_2, we merge T_3 and the query set T_{11} to form $\{T_3, T_{11}\}$, and T_{10} and the query set T_6 to form $\{T_{10}, T_6\}$ via complete SIJ; the output of $\{T_3, T_{11}\}$ will be on the order of 100 tuples, so by using gf_6, we first merge T_2 and T_4 via OU, then by using gf_5, we merge T_1 and $\{T_2, T_4\}$ via left OJ. Now we have three queries with the execution time of 57.2. Furthermore, by using gf_5, we can form $\{T_3, T_{11}, T_2, T_4\}$ via OU with execution time of 55.4. We do not merge $\{T_3, T_{11}, T_2, T_4\}$ with $\{T_3, T_{11}\}$, by using gf_6.

We run our test query sequences mainly to determine the effect of the boundary parameter values such as using 0.3 for bf, when $|R|$ is in the order of 10. If bf is set to 0.5, then the efficiency of merging dramatically increases. Although using left OJ and OU can provide improvement in the execution time, the best results are obtained in the existence of browse pattern correlation and by using SIJ.

9 Discussion and Conclusion

In this project, we restated the problem of determining "how much", and "when" to prefetch as "which queries", and "how" to merge. We assume that the query sequences of an application is known beforehand. We formalize the given query sequence as a query set and analyze the processing strategies for a given query set in three categories:

- process each query individually,
- process all the queries together, and
- partition the query set into subsets, which we call prefetch queries, and process the queries in each subset together.

In our current settings, we focused on query response time minimization. We considered and analyzed two query-access pattern correlations: (1) Browse, (2) Input, and four query merging operators: (1)c-SIJ, (2)p-SIJ, (3)Left OJ, and (4)OU. The read-ahead queries are formed via merging the queries with merging operators that are pattern-related. We also modeled the cost function for each of the query merging operators. We generated our test cases for integrated SPECJ (a manufacturing data model) and financial services data model.

This problem has two dimensions. First dimension is the problem we are already considering, which is query merging, and the second dimension is the join order optimization problem. The underlying query optimizer is responsible for the second dimension of the problem. So in our current parameters, we focus on the parameters that is least affected by the second dimension of the problem, such as focusing only on the size of the tables used in the extra join condition, which is the result of merging SIJ or left OJ, instead focusing the size of all tables in the $from$ part of the SQL query.

Our next basic goal is to discover new guidelines or revise the existing guidelines by introducing network latency and JDBC overhead and having experiments with a realistic business applications instead of generating the query sequences by ourselves. Another consideration to take into account can be the probabilities among related queries instead of assuming that the query sequence is given. In such settings, we will generate the efficiency among potential queries that may be merged and also requested with the given probabilities.

References

[1] Sibel Adali, K.S. Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *ACM SIGMOD Conf. on management of data*, pages 137–148, 1996.

[2] Philip A. Bernstein, Shankar Pal, and David Shutt. Context-based prefetch - an optimization for implementing objects on relations. *VLDB Journal*, 9(3):177–189, 2000.

[3] A. Soydan Bilgin. Incremental read-aheads. In *Proc. ICDE/EDBT Ph.D. Workshop*, Boston, MA, March 2004.

[4] A. Soydan Bilgin. Complexity analysis of query merging problem. http://www4.ncsu.edu/ asbilgin/complexity.pdf, Unpublished manuscript, May 2005.

[5] A. Soydan Bilgin, Rada Y. Chirkova, Timo J. Salo, and Munindar P. Singh. Deriving efficient sql sequences via read-aheads, full version. In *Proc. DAWAK*, Zaragoza, Spain, September 2004.

[6] A. Soydan Bilgin, Rada Y. Chirkova, Timo J. Salo, and Munindar P. Singh. Deterministic prefetching techniques for rdbms. http://www4.ncsu.edu/ asbilgin/prefetching.pdf, Unpublished manuscript, April 2005.

[7] Ivan T. Bowman and Kenneth Salem. Optimization of query streams using semantic prefetching. In *ACM SIGMOD*, Paris, France, June 2004.

[8] Shaul Dar, Michael J. Franklin, Bjorn T. Jonsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 330–341. Morgan Kaufmann Publishers Inc., 1996.

[9] Daniela Florescu, Alon Levy, Dan Suciu, and Khaled Yagoub. Optimization of run-time management of data intensive web sites. In *Proc 25th VLDB Conf*, pages 627–638, Edinburgh, Scotland, September 1999.

[10] Laura M. Haas, Donald Kossmann, and Ioana Ursu. Loading a cache with query results. In *Proc 25th VLDB Conf*, pages 351–362, 1999.

[11] Wook-Shin Han, Yang-Sae Moon, and Kyu-Young Whang. Prefetchguide: capturing navigational access patterns

for prefetching in client/server object-oriented/object-relational dbmss. *Inf. Sci.*, 152(1):47–61, 2003.

[12] Olga Kapitskaia, Raymond T. Ng, and Divesh Srivastava. Evolution and revolutions in LDAP directory caches. In *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, Proceedings*, volume 1777 of *Lecture Notes in Computer Science*, pages 202–216. Springer, 2000.

[13] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5(1):35–47, 1996.

[14] Nils Knafla. Prefetching techniques for client/server, object-oriented database systems, 1999. http://www.dcs.ed.ac.uk/home/nk/papers/th.pdf.

[15] Tom M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[16] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611. ACM Press, 2002.

[17] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. In *Proc 17th VLDB Conf*, pages 255–264, Barcelona, Spain, 1991.

[18] Dazhi Wang and Junyi Xie. An approach toward Web caching and prefetching for database management systems, 2001. www.cs.duke.edu/ junyi/cps216/report.pdf.

[19] Qingsong Yao and Aijun An. Sql-relay: An event-driven rule-based database gateway. In *WAIM*, Chengdu, China, August 2003.

[20] Qingsong Yao and Aijun An. Characterizing database user's access patterns. In *DEXA*, Zaragoza, Spain, September 2004.

A Algorithms

if $pattern\_indicator == 1$ **then**
    A randomly chosen individual query is used to build the select, from, and, where part of the merged query set because the queries in the set only differ by the value of the input parameter
    Randomly choose j, which is the index of the query q_j from the query set of $BP_{i,column\_index,index}.query\_list$;
end
$i = lhs$;
if *SIJ AND pattern_indicator* $== 1$ **then**
    if *complete SIJ* **then**
        $select_Q = select_i \bigcup SELECT_j$
    else
        partitioned SIJ $select_Q = SELECT_j$
    end
    $from_Q = from_i \bigcup from_j$;
    if $from_j \subseteq from_i$ **then**
        $where_Q = where_i$;
    else
        $where_Q = where_i + where_j$;
        foreach $\langle column\_index, index \rangle$ *pairs* **do**
            $tname\_j, cname\_j, tname\_i, cname\_i$ are temporary variables used to add a new join predicate to the SQL of the merged query $tname\_j$= $BP_{i,column\_index,index}.table\_name$;
            $cname\_j$= $BP_{i,column\_index,index}.column\_name$;
            $tname\_i$= $O_i^{column\_index}.table\_name$;
            $cname\_i$= $O_i^{column\_index}.column\_name$;
            $where_Q = where_Q + tname\_i.cname\_i = tname\_j.cname\_j -$ Selection predicates that contain $tname\_j.cname\_j$;
        end
    end
    $q_{merged} = select_Q + from_Q + where_Q$;
end
if *Left OJ AND pattern_indicator* $== 1$ **then**
    $select_{i,j} = select_i \bigcup select_j$;
    $inlineView_i = select_i + from_i + where_i$;
    foreach $\langle column_index, index \rangle$ *pairs* **do**
        $tname\_j$= $BP_{i,column\_index,index}.table\_name$;
        $cname\_j$= $BP_{i,column\_index,index}.column\_name$;
        if $select_j$ *does not contain* $tname\_j.cname\_j$ **then**
            $inlineView_j.select = inlineView_j.select \bigcup tname\_j.cname\_j$;
        end
    end
    $inlineView_j = inlineView_j.select + from_j + where_j$;
    $from_{i,j} = inlineView_i$ AS $name_i$ + LEFT OUTER JOIN $inlineView_j$ *AS* $name_j$ ON;
    foreach $\langle column_index, index \rangle$ *pairs* **do**
        $cname\_j$= $BP_{i,column\_index,index}.column\_name$;
        $cname\_i$= $O_i^{column\_index}.column\_name$;
        $from_{i,j} = from_{i,j} + name_i.cname\_i = name_j.cname\_j$;
    end
    $q_{merged} = select_{i,j} + from_{i,j}$;
end

if OU **then**

$j = rhs$;

$select_{i,j}$=$query\_identifier$ + $(select_i \bigcup select_j)$;

match $select_i$ with $select_{i,j}$ and put the matched columns in ou_select_i;

For unmatched columns, put $NULL$ in ou_select_i;

match $select_j$ with $select_{i,j}$ and put the matched columns in ou_select_j;

For unmatched columns, put $NULL$ in ou_select_j;

q_{merged}= $ou\_select_i + from_i + where_i$ UNION ALL $ou\_select_j + from_j + where_j$;

end

Algorithm 1: $merge(lhs, rhs, pattern\_indicator, op\_type)$