Deriving Efficient SQL Sequences via Read-Aheads

A. Soydan Bilgin[1], Rada Chirkova[1], Munindar Singh[1], Timo Salo[2]

[1] North Carolina State University
Raleigh NC 27695-7535, USA
[2] IBM ???
Raleigh NC , USA

Abstract. In spite of the advances in caching, query optimization, and object persistence techniques in the past few years, the cost of interactions of large-scale data-intensive applications with a relational database where the persistent objects are implemented remains a performance bottleneck. To reduce the cost of such interactions, we present a read-ahead scheme, which allows the application to reduce the number of database roundtrips by retrieving the data before it is actually needed by the transactions in the applications.
Our goal is to design generic rules for determining the efficient sequences of SQL statements for read-ahead queries on relational databases, such that the rules would be useful across application domains and data-access patterns. This paper focuses on our research methodology for generating generic access patterns and studying the parameters that influence the costs of various combinations of read-ahead SQL statements that implement the generic access patterns of applications. We explain how we model the data accesses as a directed graph and how we elaborate the efficient sequences of read-ahead SQL statements for a realistic financial domain transactions.

1 Introduction

The objective of this project is to develop new methods for improving a range of performance metrics in modern relational databases. The main direction of the project is to design efficient scalable techniques for increasing the throughput of database accesses and minimizing the cost of database interactions, by reading ahead of time the data before it is actually needed by the transactions in the data-intensive applications.

Relational databases are the most popular and commonly used DBMSs in the enterprise, so minimizing the cost of interactions between large-scale data-intensive applications and relational databases will result in considerable gains in performance of applications. Each data access operation results in one or several relatively expensive physical disk accesses and network overhead that reduce the total throughput in terms of business transactions performed in a unit of time.

Traditional caching and prefetching techniques are heavily used to efficiently handle database queries, which process large number of data objects or Web documents [1] [2]. The technique of prefetching refers to the process of guessing an application's future requests for data and getting those data into the cache before they are actually referenced. Caching is used to store the actively referenced data objects, thereby avoids unnecessary requests to the database. However, even with caching, if an access to a non-cached data item (e.g., uncacheable data, compulsory misses) occurs that entails a round-trip to the database, performance will suffer. [3] provides a feeling for the performance penalty of relational databases with and without the technique of prefetching. According to the experiments described in [3], the retrieval time of data is up to 5.5 times faster to get rows in a batch of 100 rows than a row at a time. Simple prefetch mechanisms, which create read-ahead threads for certain queries that return large quantities of data sequentially from a single table, are already used in commercial data servers or application servers.

The prefetching technique of reading ahead of time, which we refer to as read-aheads in the rest of the paper, is used to reduce the number of database roundtrips either by augmenting the current query to include the answer to the next queries or to generate additional queries that are likely to come next. In current application servers, some read-ahead techniques are already used by object managers or containers. However, these techniques should be used systematically, because incorrect guessing reduces the throughput of the server, and the overhead examining the query for possible concatenations increases the latency. Current object managers that are responsible for accessing the database cannot efficiently benefit from the application's context descriptions. In this respect, we want to help such object managers to maximize the probability of correct guessing and to have efficient database interactions by finding the rules of thumb regarding what strategies to use to determine efficient sequences of SQL statements. We want to provide a systematic vision for state-of-the-art read-ahead techniques.

2 Motivation and Problem Formulation

Our goal is to bring the potentially accessible data to application's environment in the most efficient way in terms of maximal throughput and minimal database interaction cost. We use relational structure of data such as table relationships and data dependencies, and access patterns of the applications to predict the potentially accesssible data that has the lowest retrieval cost. We separate this original problem into two parts.

1. Design generic rules for determining the efficient sequences of SQL statements for read-ahead queries on relational databases, such that the rules would be useful across application domains and data-access patterns.
2. Develop efficient and scalable algorithms for fine-tuning the read-ahead access rules in each particular application.

We seek rules of thumb that would be applicable accross application domains and data-access patterns. To derive such *generic domain-independent rules*, we need to study the parameters that influence the costs of various combinations of read-ahead SQL statements that implement the generic access patterns. These generic domain-independent rules can then fine-tuned according to the data-access patterns of particular applications.

In the first phase of our project, we look for break-even points for efficient sequences of SQL statements by discovering new parameters and by exploring known parameters such as the size of data in the tables (number of rows and columns), the size of the requested data, the complexity of SQL calls (e.g., number of join tables, join restrictions), presence of indexed columns, shape of the graph we are trying to retrieve (in a depth-first manner or breadth-first manner). Break-even points correspond to parameter values that are used in constructing efficient sequences of SQL statements, which give maximal throughput with minimal database interaction cost including data-retrieval time. As an outcome of this phase, we will have rules of thumb applicable across applications regarding what strategies to use for efficient sequences of SQL statements. At present, we don't consider network latency and we don't consider the caching aspects of the problem to answer subsequent queries from the results that were prefetched. In addition to these, we don't consider read-aheads for batch queries. For example, instead of focusing on queries such as *find the name of all customers*, we focus on queries such as *find the name of the customer with given ID* .

Example 2.1 We consider a simple financial services data model for our examples [4]. According to this model, there can be many banks, and each customer can have multiple accounts in different banks. Relations *Person* and *Organization* store detailed information about customers of the banks. The relation *Customer* stores the common attributes of persons and organizations. The relation *Account* stores account information for customers in various banks. In Transaction 1, we consider a customer representative who issues three data-access requests for the same customer from the database (The data-access requests are abbreviated with uppercase letters).

Transaction 1:

A Find the city and SSN of (person) customer '0011111'.
B Find the total amount of money that customer '0011111' has in all his accounts.
C Find the routing numbers of the banks where customer '0011111' has accounts.

For Transaction 1, request **A** requires two joins on tables for Customer, Person and Address without any read-aheads as in Figure 1. With read-aheads, it will be a smart choice to also bring data from Account table as in Figure 2, because request **B** that comes after **A**, requires data in the Account table. So using just one SQL statement with three joins as in Figure 2, we answer the first two requests **A** and **B**. For request **C**, we need to access the Bank_Customers table, so we need another SQL statement to fetch the data. As a result, with two SQL statements as in Figure 2 (three joins for the first statement and no joins for the second statement), we are able to answer the requests in Transaction 1. As

```
select Address.city, Person.ssn
from Customer, Person, Address
where Person.customerId=Customer.customerId
and Customer.addressId=Address.addressId
and Customer.customerId= '0011111'
```

Fig. 1. SQL statement for Transaction 1: A

```
AB:                                                    C:
select  Person.ssn, Address.city, sum(amount)           select routingNumber
from Customer, Account, Person, Address                 from Bank_Customers
where Person.customerId=Customer.customerId             where customerId='0011111'
and Customer.addressId=Address.addressId
and Customer.customerId=Account.customerId
and Customer.customerId= '0011111'
group by Person.ssn, Address.city
```

Fig. 2. SQL statements for Transaction 1: AB and C

```
select Address.city, Person.ssn, Bank_Customers.routingNumber, sum(amount)
from Customer, Account, Person, Address, Bank_Customers
where Person.customerId=Customer.customerId and Customer.addressId=Address.addressId
and Customer.customerId=Account.customerId and Customer.customerId= '0011111'
and Customer.customerId=Bank_Customers.customerId
group by Person.ssn, Address.city, Bank_Customers .routingNumber
```

Fig. 3. SQL statement for Transaction 1: ABC

another option, we can bring all the graph data in just one SQL statement by joining all the required tables at once; but in this case we will have a more complex query shown in Figure 3. This example shows the trade off between some of the parameters, such as the cost of the join operation versus the number of database roundtrips. Although the SQL statement in Figure 3 requires one database roundtrip, we may choose to use two SQL statements as in Figure 2 because these statements need fewer join operations in total and thus may result in lower total roundtrip time for large databases.

As the above example illustrates, our goal is not to find the minimal number of data-access statements, but to find the efficient number of simplest data-access statements. For example, one very complex data-access statement can bring all the data by reading ahead of time at one roundtrip, but this statement may not result in the efficient response time due to the cost of the join operations. Also this data-access statement can bring the data that may never be needed by the application.

3 Related Work

In [5], various types of prefetching are characterized according to its short-term and long-term benefits. In short-term prefetching, future accesses to data are predicted according to the cache's recent access history. In long-term prefetching, global object access patterns are used to identify valuable objects that are worth prefetching (i.e., if an object is accessed by one client, it is likely that it will be accessed by other clients) [5]. Both prefetching techniques are mainly used in reducing the latency used in loading web pages [6]. In our work, we use long-term prefetching that also uses data-access costs to identify cheap and valuable prefething sequences.

Prefetching is used to either pre-load the data needed for the subsequent queries for the given workload or load a specific collection of objects related to the requested object. The former case requires a more complex mechanism to track the cache contents and an analysis of which parts of the previous query is contained in the subsequent query, is required. [7] addresses the former case as an optimization for computing overlapping queries that generate Web pages (e.g., online shopping, where users narrow down their search space as they navigate through a sequence of pages). Predicate-based caching also serves the same idea where current cache content is used to answer future queries [8]. The approach in [9] uses the transition probabilities for each query to find the most probable query that can appear after the current query. So while executing the current query, they also execute the most probable subsequent query by using probable parameters and query pattern. In our approach, we take into account the cost of the data-access statements and parameters that affect this cost, to find the efficient sequence SQL statements.

[10] proposes to use a predictive cache to recognize and exploit access patterns for applications by incorporating prefetching mechanism with cache replacement mechanism to eliminate erroneous or least-likely prefetches. [11] also aims to 'pre-cache' the objects that are likely to be subsequently accessed by the application. Haas et al. propose a heuristic approach to cache and prefetch the objects whose object identifiers were returned as part of the query, so they make prefetching decisions according to the object identifiers found in the result set of the query. However, instead of focusing on finding the most beneficial prefetches, their goal is to find the cost of caching the prefetched objects. They incorporate the prefetching process into query processing to find the best execution plan by considering the cost of caching the prefetched tuples.

[3] specifially addresses the prefetching technique on relational databases where persistent objects are implemented. They use the context of an object as a predictor for future accesses in navigational applications. This context describes the structure in which the object was fetched. Main prefetching methods are listed (e.g., prefetching all the attributes of the requested object(s)). The results are applicable accross application domains, because they use generic access patterns that are applicable across a wide range of applications. However, they only make one-level prefetching for referenced objects, so they don't actually answer the 'how deep' question for read-aheads. Their overall goal is to minimize database latency for future data-access statements, so they don't explore the cost of efficient sequence of data-access statements.

We explore the cost of data-access statements to find efficient generic data-access patterns. None of the previous work consider the prefetching problem both with query optimization parameters and navigational access patterns, such as following a relationship, at the same time. Also by providing a mechanism for merging simple data-access statements to find an efficient sequence of data-access statement, we actually use a different aspect for multi-query optimization [12] where dependencies or common subexpressions between the queries in a sequence are explored and computed.

4 Proposed Approach

Applications use objects, but these objects are mapped to tuples of the appropriate tables. In relational databases, the objects accessed by the applications are always associated with each other. Most of the time, the data in the database is accessed according to these associations, and again most of the time these associations are intuitive and work just as you would expect. This is an important observation for the first phase of our project in which we don't specifically use data-access patterns of particular applications. Instead, for the first phase of the project, our goal is to come up with generic read-ahead rules that are applicable across applications. An important property of applications for systems such as health-care, financial, or human resources is repetitive usage of the same query templates. 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. The similar associations and dependencies that can be found in different domains form a basis for guessing the useful generic access patterns in our project.

We generate SQL statements that implement access patterns, which are part of the given access sequence. To generate various combinations of SQL statements for the given access sequence with read-ahead functionality, we can use the following prefetch methods:

– Prefetch only primary keys of the associated objects
– Prefetch primary and non-primary foreign keys
– Prefetch key and non-key attributes, or only non-key attributes
– Prefetch via traversing the inheritance-extension, one-to-many association (i.e., aggregation), or many-to-many type of relationships

Generic access patterns aren't enough to determine the most beneficial and efficient read-ahead scheme configuration. These patterns are helpful to determine the useful sequence of SQL statements. On the other hand, we need to find efficient sequences of (merged) SQL statements. Finding such efficient sequences includes answering the following questions:

1. How much to read ahead? This question requires figuring out which tables and table columns may be subsequently accessed, and how the structure of data (e.g., existence of an indexed attribute) and relational constraints affect the structure of a read-ahead query.

2. How deep to read ahead? This question requires figuring out the levels of the object hierarchy that may be subsequently accessed. How does the number of joins affect the efficiency of sequence of SQL statements?
3. In what direction to read ahead? In each level of the object hierarchy, each object can be associated with many different objects. This question requires figuring out the trade-off of typical directions of the traversal on the object hierarchy that is stored in the relational database.

5 Research Methodology and Experiments

We construct a testbed to experiment with the parameters that affect the cost of SQL statements used while reading ahead of time. The focus of our case study is to discover and experiment with the data-access patterns of financial applications with parameters that are related to the structure of the data in the database. For this case study, we use a slightly modified version of the standard data model for financial services (e.g., banking and investment services) as described in [4]. Some of the main entities of this model are shown in Figure 4. We use Oracle 9i as our data server to implement the entire data model which has 55 tables and maximum fan-out of 7 relationships. After creating a sample database for this model, we listed the possible SQL queries that can be issued by the applications. We modeled all the frequent data accesses in one directed graph. The Figure5 shows such a sample graph with six SQL statements. In this directed graph, vertices correspond to simple SQL statements and edges correspond to average first-order transition probabilities between the statements. These probabilities can be calculated and updated using access log files of the database server. Right now they are generated randomly. Based on this simple hypothetical transition graph, whenever A is accessed there is a 40% chance that B will be accessed next. We assume an acylic graph that can have multiple sources and sinks for the first phase of the project.

We need to merge simple data-access statements to find optimal sequences of SQL statements for the given access sequence. There are two principle ways to combine two or more SQL statements. Either the statements are executed *at the same time* or they are executed *one after the other*. We call the first combination *merged execution* and denote it by operator $*$; the second combination represents *sequential execution* and is denoted by operator +. We can apply $*$ repeatedly to describe access patterns, and apply + repeatedly to combine patterns to form the access sequence. By definition, $*$ is commutative, while + is not, and $*$ has precedence over +. For example, if we have an access sequence *ABEF* for the transaction in Figure 6, we can come up with SQL sequences such as $A*B*E+F$, $A*B*E*F$, $A*C*B+E*F$, $A*B+E*F$, $A+B*E*F$ and so on. Figure 7 represents the level by level formation of read-ahead SQL sequences as a directed graph. In this graph, each node represents the set of SQL statements that can be executed at the same time or one after the other. We chose this representation, otherwise the number of nodes will have exponential complexity.

Let the cost of evaluating a SQL statement be defined as *cost(Q)* = Response

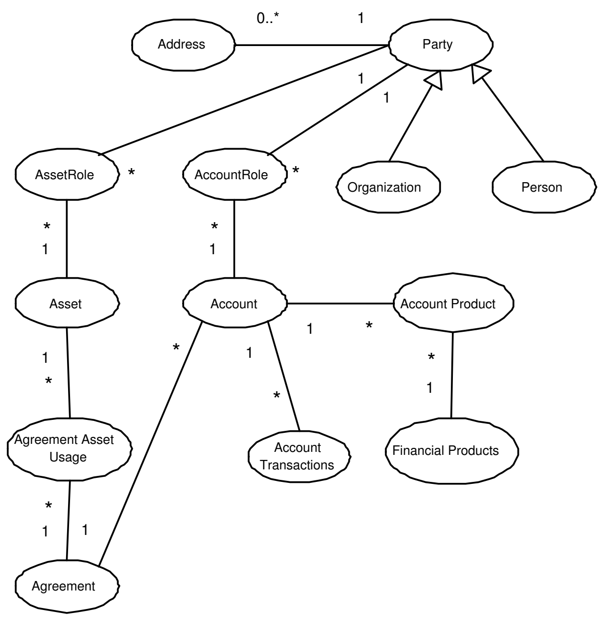

Fig. 4. Some entities in the financial services model

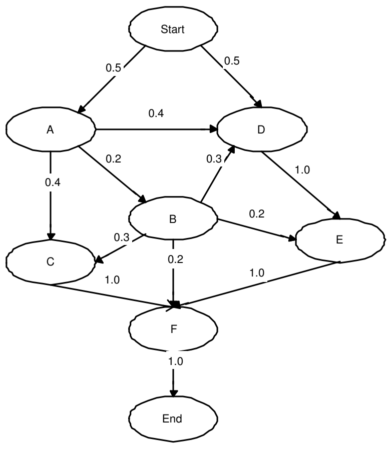

Fig. 5. A Query transition graph

Time of Q and the efficieny of a read-ahead SQL sequence is the fraction of the minimal cost of the previous level read-ahead SQL sequences to the cost of the current sequence. The previous level read-ahead SQL sequences are found by applying dynamic programming algorithm (matrix-chain-order algorithm) on the current node. For example, the efficieny of read-ahead sequence $A*B*E*F$ is the fraction of min(*cost(A+B*E*F, A*B+E*F, A*B*E+F)*) to *cost(A*B*E*F)*. Our goal is to find the read-ahead SQL sequences that have efficieny close to 1.0 by estimating the cost of the read-ahead SQL sequences by using the rules, which we want to explore. If the efficiencies of the previous level read-ahead sequence are all smaller than 1.0, then we should stop exploring down the graph to find more efficient read-ahead SQL sequences.

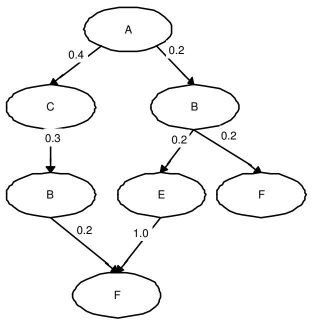

Fig. 6. A transaction's data flow graph

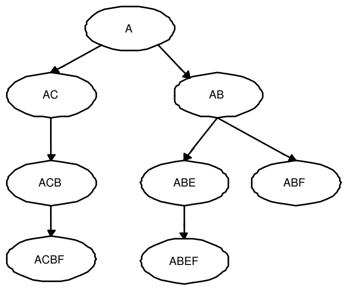

Fig. 7. The read-ahead scenarios graph for Figure7

By testing the possible read-ahead scenarios for the SQL statements in Figure 5, which are also listed in Section 8, we can come up with multiple meaningful application logic. Here, we list the meanings of these SQL statements (X and Y are variable names).

A Find the IDs and balance of the accounts that customer X 'owns'
B Find the IDs and balance of the accounts, where another account Y uses as 'overdraft' account(s)
C Find the account IDs and balance of the customer, who is 'co-owner' of the account Y
D Find the IDs, feature name, and the value of the products that account Y is linked
E Find the IDs and amount of the transactions on account Y
F Find the agreement IDs, the parties and the roles of these parties that are linked to these agreements, where account Y is linked via an ownership or co-ownership relation

The above list can be easily extended by using more complex query transition graphs, but these simple SQL statements is enough to derive complex and meaningful read-ahead scenarios, which help us in determining rules of thumb regarding what strategies to use for efficient sequences of read-ahead SQL statements.

//ADD TABLE HERE

During our experiments, we played with the cardinality of the tables as 10K, 100K, 1000K, the cardinality of returned data_set as 1,1K,10K, the number of the common joined tables and common join restrictions via primary or foreign keys in successive queries. We basically encountered with the following situations in our first set of experiments:

– The benefit of the existence or the number of joined tables in successive queries is observed for merged SQL statements *A*B* and *A*B*C*. Although in both cases, the efficieny of merged statements was smaller than `1.0`, the efficiency can be bigger than `1.0` with the addition of network latency as a cost factor for huge size of returned data_set. With small table sizes we get slightly better efficiencies for *A*B*C* than for *A*B*. However, this advantage disappeared when we increase the table size from 100K to 1000K.
– The efficieny of *A*B* is not affected by the cardinality of common joined table (*Account*), but it decreases if we increase the cardinality of the tables that are uncommon in *A* and *B*. Also the efficieny decreases with the increase in the number of the returned tuples that was selected by the selection predicates or join restriction via keys. For this situation, for relations *R* and *S*, the selection predicates such as *R.key=1* and *S.key=1* have more negative impact than *R.key=S.key*.
– For *A*B*F*, its efficieny is much worse than *A*B+F*, although both three tables have one common table. Here, the main problem is the size of the junk-data that returns as a result of join operations. This inefficieny is especially affected by the cardinality of the result data-set of *F*.
– If two succesive queries have no common tables, then they shouldn't be merged.
– For merged queries that include 1-to-many relationships such as in *B*F*, the increase in returned data cardinality has more negative impact on the efficiency than the cardinality of the tables.

6 Discussion

The results of our work can be used to effectively preload associated dataset of a requested object that may be subsequently accessed. Our work can also be integrated with query optimizers to find the efficient execution plans for compound SQL statements. This integration provides us another aspect for multi-query optimization.

This work can also be integrated with Container Managed Persistence containers or Java Data Object drivers as a performance tuning technique. Using this technique, we will be able to get ahead of time the working set of objects for the current transaction in very concurrent environments. Also this work can be helpful in determining the right cache size for systems that use prefetching.

One of the important implementation challenges is to merge simple SQL statements on-the-fly to experiment with the compound SQL statements. This isn't a trivial task, because it requires the detailed knowledge of the relational structure of data. Even for the statements that are syntactically similar, we can have different merged SQL statements.

We use applications' common behaviours and the cost of database interactions to generate read-ahead rules that can provide a significant performance gain for systems where many concurrent data-intensive read-only applications access huge databases.

In the near future, we want to take into account the network latency, and complex index structures for our rules. We also want to find the threshold of the transition probabilities to use in the read-ahead scenarios efficieny formula. In addition to these, in subsequent phases of the project, we will explore the effect of object-to-relational mapping technique on our generic domain-independent rules and will develop and test learning algorithms to increase the efficieny of our generic rules for the data-access patterns of particular applications.

7 Acknowledgements

This research is supported under NCSU CACC Grant 11019.

References

1. Kroeger, T.M., Long, D.D.E., Mogul, J.C.: Exploring the bounds of web latency reduction from caching and prefetching. In: USENIX Symposium on Internet Technologies and Systems. (1997)
2. Adali, S., Candan, K., Papakonstantinou, Y., Subrahmanian., V.: Query caching and optimization in distributed mediator systems. In: ACM SIGMOD Conf. on management of data. (1996) 137–148
3. Bernstein, P.A., Pal, S., Shutt, D.: Context-based prefetch - an optimization for implementing objects on relations. VLDB Journal **9** (2000) 177–189
4. Silverston, L.: The Data Model Resource Book. Volume 2. John Wiley and Sons, New York (2001)
5. Venkataramani, A., Yalagandula, P., Kokku, R., Sharif, S., Dahlin, M.: The potential costs and benefits of long term prefetching for content distribution. In: Proc of Web Content Caching and Distribution Workshop. (2001)
6. Davison, B.D.: The Design And Evaluation Of Web Prefetching and Caching Techniques. PhD thesis, Department of Computer Science, Rutgers University (2002)
7. Florescu, D., Levy, A., Suciu, D., Yagoub, K.: Optimization of run-time management of data intensive web sites. In: Proc 25th VLDB Conf, Edinburgh, Scotland (1999) 627–638
8. Keller, A.M., Basu, J.: A predicate-based caching scheme for client-server database architectures. VLDB Journal **5** (1996) 35–47
9. Wang, D., Xie, J.: An approach toward Web caching and prefetching for database management systems (2001) www.cs.duke.edu/ junyi/cps216/report.pdf.
10. Palmer, M., Zdonik, S.B.: Fido: A cache that learns to fetch. In: Proc 17th VLDB Conf, Barcelona, Spain (1991) 255–264
11. Haas, L.M., Kossmann, D., Ursu, I.: Loading a cache with query results. In: Proc 25th VLDB Conf. (1999) 351–362

12. Choenni, R., Kersten, M., Saad, A., Akker, J.: A framework for multi-query optimization. In: Proc. COMAD 8th Int. Conference on Management of Data. (1997) 165–182

8 Appendix

```
A:
select a.account_number, a.balance
from Account a, Account_Role_Type art, Account_Role al
where art.name=' owner' and
art.role_type_id=al.role_type_id and
al.party_id=X and al.account_number=a.account_number;

B:
select a.account_number, a.balance
from Account a, Account_Relation_Type art, Account_Relation ar
where art.name=' overdraft' and
art.account_relation_type_id=ar.account_relation_type_id and
ar.account_number_to=Y and ar.account_number_from=a.account_number;

C:
select a.account_number, a.balance
from Account a, Account_Role_Type art, Account_Role_Type art2, Account_Role al, Account_Role al2
where art.name=' co-owner' and
art.role_type_id=al.role_type_id and
al.account_number=Y and al.party_id=al2.party_id and
art2.role_type_id=al2.role_type_id and art2.name=' owner' and
al2.account_number=a.account_number;

D:
select pfa.product_id, pf.feature_name, pfa.value
from Account_Product ap, Product_Feature pf, Prod_Feature_Applicable pfa
where ap.account_number=Y and
ap.produc_id=pfa.product_id and
pf.prod_feature_id=pfa.prod_feature_id ;

E:
select a.account_trans_id, a.amount
from Account_Transaction a, Account_Trans_Type att
where a.account_number=Y and att.name=' ' purchase' and
att.account_trans_type_id=a.acount_trans_type_id;

F:
select ar.agreement_id, ar_party_id, ar.role_type_id
from Agreement_Role_Type art, Agreement_Role ar, Account a
where a.account_number=Y and a.agreement_id=ar.agreement_id and
(art.description=' owner' or art.description=' co-owner' ) and art.role_type_id=ar.role_type_id;
```

Fig. 8. Queries for Figure 5

This article was processed using the LaTeX macro package with LLNCS style