Integrating Association Rule Mining with Relational Database Systems Alternatives and Implications

Sunita Sarawagi
Shiby Thomas
Rakesh Agrawal

IBM Research Division
Almaden Research Center
Harry Road
San Jose CA

LIMITED DISTRIBUTION NOTICE

This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication It has been issued as a Research Report for early dissemination of its contents In view of the transfer of copyright to the outside publisher its distribution outside of IBM prior to publication should be limited to peer communications and speci c requests After outside publication requests should be lled only by reprints or legally obtained copies of the article e g payment of royalties

Integrating Association Rule Mining with Relational Database Systems Alternatives and Implications

Sunita Sarawagi
Shiby Thomas
Rakesh Agrawal

IBM Research Division
Almaden Research Center
Harry Road
San Jose CA

ABSTRACT Data mining on large data warehouses is becoming increasingly important In support of this trend we consider a spectrum of architectural alternatives for coupling mining with database systems These alternatives include loose coupling through a SQL cursor interface encapsulation of a mining algorithm in a stored procedure caching the data to a le system on the y and mining tight coupling using primarily user de ned functions and SQL implementations for processing in the DBMS We comprehensively study the option of expressing the mining algorithm in the form of SQL queries using Association rule mining as a case in point We consider four options in SQL and six options in SQL enhanced with object relational extensions SQL OR Our evaluation of the di erent architectural alternatives shows that from a performance perspective the Cache Mine option is superior although the performance of the SQL OR option is within a factor of two Both the Cache Mine and the SQL OR approaches incur a higher storage penalty than the loose coupling approach which performance wise is a factor of to worse than Cache Mine The SQL implementations were too slow to qualify as a competitive option We also compare these alternatives on the basis of qualitative factors like automatic parallelization development ease portability and inter operability As a byproduct of this study we identify some primitives for native support in database systems for decision support applications

Current a liation Dept of Computer Information Science Engineering University of Florida Gainesville

Introduction

An ever increasing number of organizations are installing large data warehouses using relational database technology There is a huge demand for mining nuggets of knowledge from these data warehouses

The initial research on data mining was concentrated on de ning new mining operations and developing algorithms for them Most early mining systems were developed largely on le systems and specialized data structures and bu er management strategies were devised for each algorithm Coupling with database systems was at best loose and access to data in a DBMS was provided through an ODBC or SQL cursor interface e g Int AAB HFK IM

Researchers of late have started to focus on issues related to integrating mining with databases There have been language proposals to extend SQL to support mining operators For instance the query language DMQL HFK extends SQL with a collection of operators for mining characteristic rules discriminant rules classi cation rules association rules etc The M SQL language IVA extends SQL with a special uni ed operator Mine to generate and query a whole set of propositional rules Another example is the mine rule MPC operator for a generalized version of the association rule discovery problem Query o cks for association rule mining using a generate and test model has been proposed in TAC

The issue of tightly coupling a mining algorithm with a relational database system from the systems point of view was addressed in AS b This proposal makes use of user de ned functions UDFs in SQL statements to selectively push parts of the computation into the database system The objective was to avoid one at a time record retrieval from the database saving both the copying and process context switching costs In SK the focus is on developing a mining system with minimal database interaction The SETM algorithm HS for nding association rules was expressed in the form of SQL queries However as shown in AMS SETM is not e cient and there are no results reported on running it against a relational DBMS Recently the problem of expressing the association rules algorithm in SQL has been explored in RIC We discuss this work later in the paper

Goal

This paper is an attempt to understand implications of various architectural alternatives for coupling data mining with relational database systems In particular we are interested in studying how competitive can a mining computation expressed in SQL be compared to a specialized implementation of the same mining operation

There are several potential advantages of a SQL implementation One can make use of the database indexing and query processing capabilities thereby leveraging on more than a decade of e ort spent in making these systems robust portable scalable and concurrent One can also exploit the underlying SQL parallelization particularly in an SMP environment The DBMS support for checkpointing and space management can be valuable for long running mining algorithms

The architecture we have in mind is schematically shown in Figure We visualize that the desired mining operation will be expressed in some extension of SQL or a graphical language A preprocessor will generate appropriate SQL translation for this operation We consider translations that can be executed on a SQL MS relational engine as well as translations that require some of the newer object relational capabilities

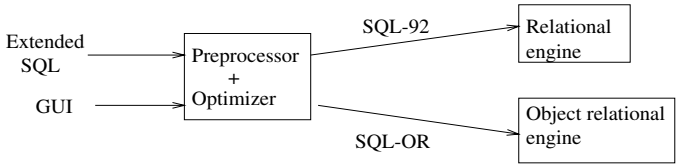


Figure SQL architecture for mining in a DBMS

being designed for SQL Kul Speci cally we assume availability of blobs user de ned functions and table functions PR We do not assume mining speci c extensions in the underlying relational engine however identi cation of such extensions is a secondary goal of this study

We compare the performance of the above SQL architecture with the following alternatives

Read directly from DBMS Data is read tuple by tuple from the DBMS to the mining kernel using a cursor interface Data is never copied to a le system We consider two variations of this approach One is the loose coupling approach where the DBMS runs in a di erent address space from the mining process This is the approach followed by most existing mining systems A potential problem with this approach is the high context switching cost between the DBMS and the mining process AS b In spite of the block read optimization present in many systems e g Oracle Ora DB Cha where a block of tuples is read at a time the performance could su er The second is the stored procedure approach where the mining algorithm is encapsulated as a stored procedure Cha that runs in the same address space as the DBMS The main advantage of both these approaches is greater programming exibility and no extra storage requirement Also any previous le system code can be easily transferred to work on data stored in the DBMS The mined results are stored back into the DBMS

Cache mine This option is a variation of the Stored procedure approach where after reading the entire data once from the DBMS the mining algorithm temporarily caches the relevant data in a lookaside bu er on a local disk The cached data could be transformed to a format that enables e cient future accesses The cached data is discarded when the execution completes This method has all the advantages of the stored procedure approach plus it promises to have better performance The disadvantage is that it requires additional disk space for caching Note that the permanent data continues to be managed by the DBMS

User de ned function UDF The mining algorithm is expressed as a collection of user de ned functions UDFs Cha that are appropriately placed in SQL data scan queries Most of the processing happens in the UDF and the DBMS is used primarily to provide tuples to the UDFs Little use is made of the query processing capability of the DBMS The UDFs are run in the unfenced mode same address space as the database Such an implementation was presented in AS b The main attraction of this method over Stored procedure is performance since passing tuples to a stored procedure is slower than passing it to a UDF Otherwise the processing happens in almost the same manner as in the stored procedure case The main disadvantage is the development cost since the entire mining algorithm has to be written as UDFs involving signi cant code rewrites AS b This option can be viewed as an extreme case of the SQL OR approach where UDFs do all the processing

Methodology

We do both quantitative and qualitative comparisons of the architectures stated above with respect to the problem of discovering Association rules AIS against IBM DB Universal Server IBM We present performance results only for the Boolean association rules problem However we have also experimented with generalized association rules discovery SA and sequential patterns discovery AS SA problems and found similar results

For the loose coupling and Stored procedure architectures we use the implementation of the Apriori algorithm AMS for nding association rules provided with the IBM data mining product Intelligent Miner Int For the Cache Mine architecture we used the space option provided in Intelligent Miner that caches the data in a binary format after the rst pass For the UDF architecture we use the UDF imple mentation of the Apriori algorithm described in AS b For the SQL architecture we consider two classes of implementations one uses only the features supported in SQL and the other uses object relational extensions to SQL henceforth referred to as SQL OR We consider four di erent implementations in the rst case and six in the second These implementations di er in the way they exploit di erent features of SQL We compare the performance of these di erent approaches using four real life datasets We also use synthetic datasets at various points to better understand the behavior of di erent algorithms

Our focus in this paper is on the performance of various architectures We do not consider the issues of the language constructs required to extend SQL with mining features nor do we discuss the preprocessing step shown in Figure

One point to be kept in mind when reading the paper is that our conclusions about the comparison of di erent architectures are based on the current associations algorithms and DBMS systems DB UDB version If the algorithms are made signi cantly faster or the DBMS systems performance characteristics change drastically these conclusions may no longer be valid

Paper Layout

The rest of the paper is organized as follows In Section we cover background material where we give a brief review of the problem of nding association rules and the Apriori algorithm AMS and discuss our assumption of the input and output format In Section we present the overview of the SQL implementations In Sections and we elaborate on di erent ways of doing the support counting phase of Associations in SQL Section presents SQL implementations whereas Section gives implementations in SQL OR In Section we present a performance comparison of the di erent architectural alternatives using real life and synthetic datasets We also include qualitative comparisons of the di erent architectures along the dimensions of development and maintenance ease storage and memory requirements portability inter operability and parallelizability In Section we propose primitives in a relational DBMS that we believe would be useful for a large class of mining algorithms We conclude with a summary of results and directions for future work in Section

Background

Association Rules

Given a set of transactions where each transaction is a set of items an association rule AIS is an expression X Y where X and Y are sets of items The intuitive meaning of such a rule is that the transactions that contain the items in X tend to also contain the items in Y An example of such a rule might be that of transactions that contain beer also contain diapers of all transactions contain both these items Here is called the con dence of the rule and the support of the rule The problem of mining association rules is to nd rules that satisfy a user speci ed minimum support and minimum con dence

The association rule mining problem can be decomposed into two subproblems AIS

Find all combinations of items called frequent itemsets whose support is greater than minimum support

Use the frequent itemsets to generate the desired rules The idea is that if say ABCD and AB are frequent then the rule AB CD holds if the ratio of support ABCD to support AB is at least as large as the minimum con dence Note that the rule will have minimum support because ABCD is frequent

Apriori Algorithm

We use the Apriori algorithm AMS as the basis for our presentation There are recent proposals for improving the Apriori algorithm by reducing the number of data passes Toi BMUT They all have the same basic data ow structure as the Apriori algorithm Our goal in this work is to understand how best to integrate this basic structure within a database system Later in the paper in Section we discuss how our conclusions extrapolate to these algorithms

The Apriori algorithm for nding frequent itemsets makes multiple passes over the data In the kth pass it nds all itemsets having k items called the k itemsets Each pass consists of two phases Let F represent the set of frequent k itemsets and C the set of candidate k itemsets potentially frequent itemsets First is the candidate generation phase where the set of all frequent k itemsets F found in the k th pass is used to generate the candidate itemsets C The candidate generation procedure ensures that C is a superset of the set of all frequent k itemsets A specialized in memory hash tree data structure is used to store C Then data is scanned in the support counting phase For each transaction the candidates in C contained in the transaction are determined using the hash tree data structure and their support count is incremented At the end of the pass C is examined to determine which of the candidates are frequent yielding F The algorithm terminates when F or C becomes empty

Input output formats

Input format The input is a transaction table T with two column attributes transaction identi er tid and item identi er item For a given tid typically there are multiple rows in the transaction table corresponding to di erent items that belong to the same transaction The number of items per transaction is variable and unknown during table creation time Thus alternative schemas may not be convenient In particular assuming that all items in a transaction appear as di erent columns of a single tuple e g RIC is not practical

because often the number of items per transaction can be more than the maximum number of columns that the database supports For instance for one of our real life datasets the maximum number of items per transaction is and for another it is In contrast the corresponding average number of items per transaction is only and respectively

Output format The output is a collection of rules of varying length The maximum length of these rules is much smaller than the number of items and is rarely more than a dozen Therefore a rule is represented as a tuple in a xed width table where the extra column values are set to NULL to accommodate rules involving smaller itemsets The schema of a rule is item item len rulem con dence support where k is the size of the largest frequent itemset The len attribute gives the length of the rule and the rulem attribute gives the position of the in the rule For instance if k the rule AB CD which has con dence and support is represented by the tuple A B C D NULL The frequent itemsets are represented the same way as rules but do not have the rulem and con dence attributes

Associations in SQL

In Section we present the candidate generation procedure in SQL and in Section we present the support counting procedure Finally in Section we present the rule generation procedure

Candidate generation in SQL

Each pass k of the Apriori algorithm rst generates a candidate itemset set C from frequent itemsets F of the previous pass

Given F the set of all frequent k itemsets the Apriori candidate generation procedure AMS returns a superset of the set of all frequent k itemsets We assume that the items in an itemset are lexico graphically ordered Since all subsets of a frequent itemset are also frequent we can generate C from F as follows

In the join step a superset of the candidate itemsets C is generated by joining F with itself

```
insert into C   select I   item        I   item      I   item
from        F     I   F     I
where       I   item      I   item   and


                I   item        I   item      and
                I   item        I   item
```

For example let F be After the join step C will be Next in the prune step all itemsets c C where some k subset of c is not in F are deleted Continuing with the example above the prune step will delete the itemset because the subset is not in F We will then be left with only in C

We can perform the prune step in the same SQL statement as the join step by writing it as a k way join as shown in Figure A k way join is used since for any k itemset there are k subsets of length k for

which F needs to be checked for membership The join predicates on I and I remain the same After the join between I and I we get a k itemset consisting of I item I item I item For this itemset two of its k length subsets are already known to be frequent since it was generated from two itemsets in F We check the remaining k subsets using additional joins The predicates for these joins are enumerated by skipping one item at a time from the k itemset as follows We rst skip item and check if subset I item I item I item belongs to F as shown by the join with I in the gure In general for a join with I we skip item r Figure gives an example for k We construct a primary index on item item of F to e ciently process these k way joins using index probes

C need not always be materialized before the counting phase Instead the candidate generation can be pipelined with the subsequent SQL queries used for support counting

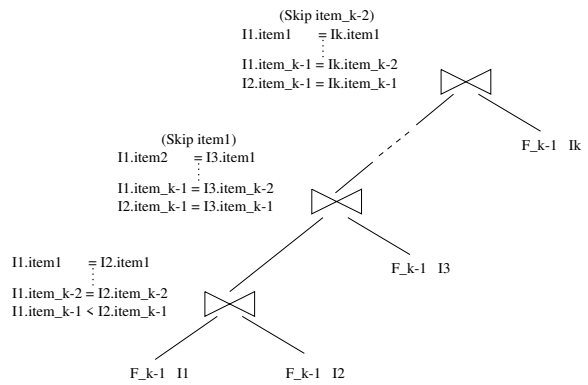


Figure Candidate generation for any

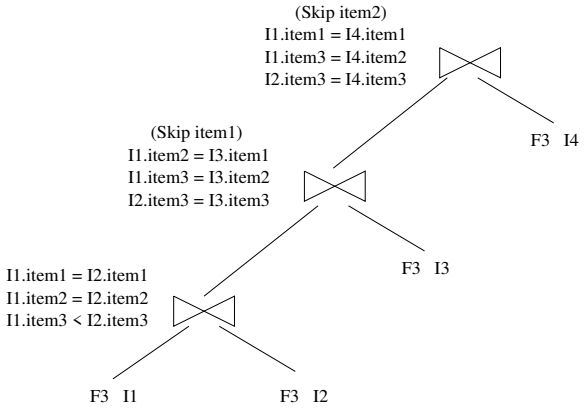


Figure Candidate generation for

Counting support to nd frequent itemsets

This is the most time consuming part of the association rules algorithm We use the candidate itemsets C and the data table T to count the support of the itemsets in C We consider two di erent categories of SQL implementations

A The rst one is based purely on SQL We discuss four approaches in this category in Section

B The second utilizes object relational extensions like UDFs BLOBs Binary large objects and table func tions Table functions PR are virtual tables associated with a user de ned function which generate tuples on the y They have pre de ned schemas like any other table The function associated with a table function can be implemented as a UDF Thus table functions can be viewed as UDFs that return a collection of tuples instead of scalar values

We discuss six approaches in this category in Section UDFs in this approach are light weight and do not require extensive memory allocations and coding unlike the UDF architectural option Section

Rule generation

In order to generate rules having minimum con dence minconf we rst nd all non empty proper subsets of every frequent itemset l Then for each subset m we nd the con dence of the rule m l m and output the rule if it is at least minconf

In the support counting phase the frequent itemsets of size k are stored in table F Before the rule generation phase we merge all the frequent itemsets into a single table F The schema of F consists of k attributes item item support len where k is the size of the largest frequent itemset and len is the length of the itemset as discussed earlier in Section

We use a table function GenRules to generate all possible rules from a frequent itemset The input argument to the function is a frequent itemset For each itemset it outputs tuples corresponding to rules with all non empty proper subsets of the itemset in the consequent The table function outputs tuples with k attributes T_item T_item T_support T_len T_rulem The output is joined with F to nd the support of the antecedent and the con dence of the rule is calculated by taking the ratio of the support values Figure illustrates the rule generation query

```
insert into R select T_item T_item
    t support T_len T_rulem t support f support
from F f table GenRules f item f item
       f len f support as t F f
where t T_item f item or t T_rulem

AND t T_item f item or t T_rulem k
AND t T_rulem f len
AND t T_support f support minconf
```

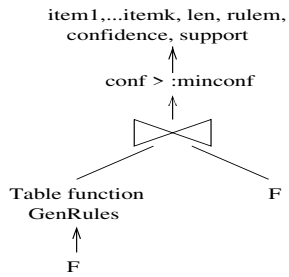


Figure Rule Generation

We can also do rule generation without using table functions and base it purely on SQL The rules are generated in a level wise manner where in each level k we generate rules with consequents of size k Further we make use of the property that for any frequent itemset if a rule with consequent c holds then so do rules with consequents that are subsets of c as suggested in AMS We can use this property to generate rules in level k using rules with k long consequents found in the previous level much like the way we did candidate generation in Section

The fraction of the total running time spent in rule generation is very small Therefore we do not discuss detailed performance evaluation of rule generation algorithms

Support counting using SQL

We studied four approaches in this category KwayJoin wayJoin GroupBy and Subquery

K way joins

In each pass k we join the candidate itemsets C with k transaction tables T and follow it up with a group by on the itemsets as shown in Figure The gure also shows a tree diagram of the query These tree diagrams are not to be confused with the plan trees that could look quite di erent

```
insert into F  select item      item   count
from       C   T t      T t
where      t  item    C  item  and

           t  item    C  item  and
           t  tid    t  tid and

           t     tid    t  tid
group by   item  item      item
having     count        minsup
```

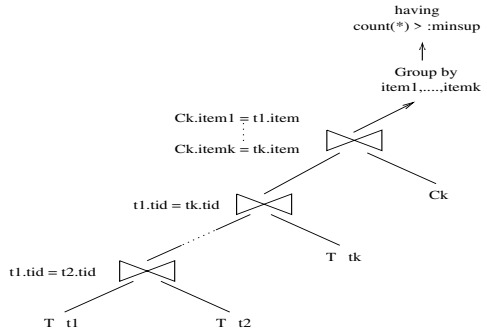


Figure Support Counting by K way join

This SQL computation when merged with the candidate generation step is similar to the one proposed in TAC as a possible mechanism to implement query o cks

For pass we use a special optimization where instead of materializing C we replace it with the way joins between the F s as shown in the candidate generation phase in section This saves the cost of materializing C and also provides early ltering of the Ts based on F instead of the larger C which is almost a cartesian product of the F s In contrast for other passes corresponding to k C could be smaller than F because of the prune step

Three way joins

The above approach requires k way joins in the kth pass We can reduce the cardinality of joins to using the following approach which bears some resemblance to the AprioriTid algorithm in AMS Each candidate itemset C in addition to attributes item item has three new attributes oid id id oid is a unique identi er associated with each itemset and id and id are oids of the two itemsets in F from which the itemset in C was generated as discussed in Section In addition in the k for k pass we generate a new copy of the data table T with attributes tid oid that keeps for each tid the oid of each itemset in C that it supported For support counting we rst generate T from T and C and then do a group by on T to nd F as follows

```
insert into T  select t  tid  oid
from       C  T     t  T     t
where      t  oid    C  id  and t  oid    C  id  and t  tid    t  tid
```

```
insert into F   select oid  item        item    cnt
from C
        select oid as cid  count      as cnt  from T
       group by oid having count          minsup  as temp
where C   oid     cid
```

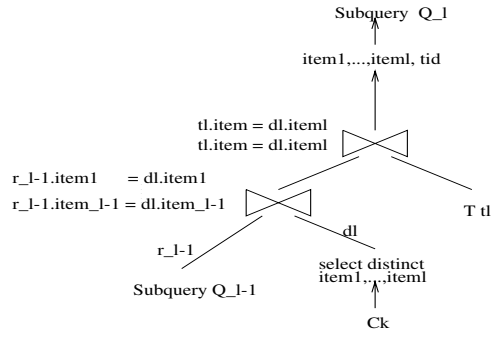
Subquery based

This approach makes use of common pre xes between the itemsets in C to reduce the amount of work done during support counting The support counting phase is split into a cascade of k subqueries The l th subquery Q see Figure nds all tids that match the distinct itemsets formed by the rst l columns of C call it d The output of Q is joined with T and d the distinct itemsets formed by the rst l columns of C to get Q The nal output is obtained by a group by on the k items to count support as above Note that the nal select distinct operation on the C when l k is not necessary

For pass the special optimization of the KwayJoin approach is used

```
insert into F   select item        item    count
    from Subquery Q    t
    group by item   item        item
    having count          minsup

Subquery Q   for any l between    and k
    select item        item    tid
    from T t    Subquery Q       as r
       select distinct item       item  from C    as d
    where r      item      d  item   and      and
          r      item         d  item    and
          r      tid     t  tid and
          t  item     d  item
```



Tree diagram for Subquery

```
Subquery Q   No subquery Q
```

Figure Support counting using subqueries

Two group bys

This approach avoids the multi way joins used in the previous approaches by joining T and C based on whether the item of a tid item pair of T is equal to any of the k items of C Then do a group by on item item tid ltering tuples with count equal to k This gives all itemset tid pairs such that the tid supports the itemset Finally as in the previous approaches do a group by on the itemset item item ltering tuples that meet the support condition For pass we apply the same optimization as in the KwayJoin approach where we avoid materializing C

Datasets	Records in millions R	T ransactions in millions T	Items in thousands I	Avg items per transaction R T
Dataset A				
Dataset B				
Dataset C				
Dataset D				

Table Description of di erent real life datasets

```
insert into F  select item      item   count
from        select item      item   count
            from   T  C
            where  item    C  item  or

                   item    C  item
            group by item       item   tid
            having count       k  as temp
group by   item        item
having     count        minsup
```

Performance comparison of SQL approaches

In this section we compare the performance of the four SQL approaches

Our experiments were performed on Version of IBM DB Universal Server installed on a R S Model with a MHz CPU MB main memory and a GB disk with a measured transfer rate of MB sec

We selected four real life datasets obtained from mail order companies and retail stores for the experiments These datasets di er in the values of parameters like the number of tid item pairs number of transactions tids number of items and the average number of items per transaction Table summarizes characteristics of these datasets

For these experiments we built a composite index item item on C k di erent indices on each of the k items of C and a tid item and a item tid index on the data table The goal was to let the optimizer choose the best plan possible We do not include the index building cost in the total time

In Figure we show the total time taken by the four approaches KwayJoin wayJoin Subquery and GroupBy For comparison we also show the time taken by the Loose coupling approach because this is the approach currently used by existing systems The graph shows the total time split into candidate generation time Cgen and the time for each pass From these set of experiments we can make the following observations

Overall the best approach in the SQL category is the Subquery approach An important reason for its superior performing over the KwayJoin approach is exploitation of common pre xes between candidate

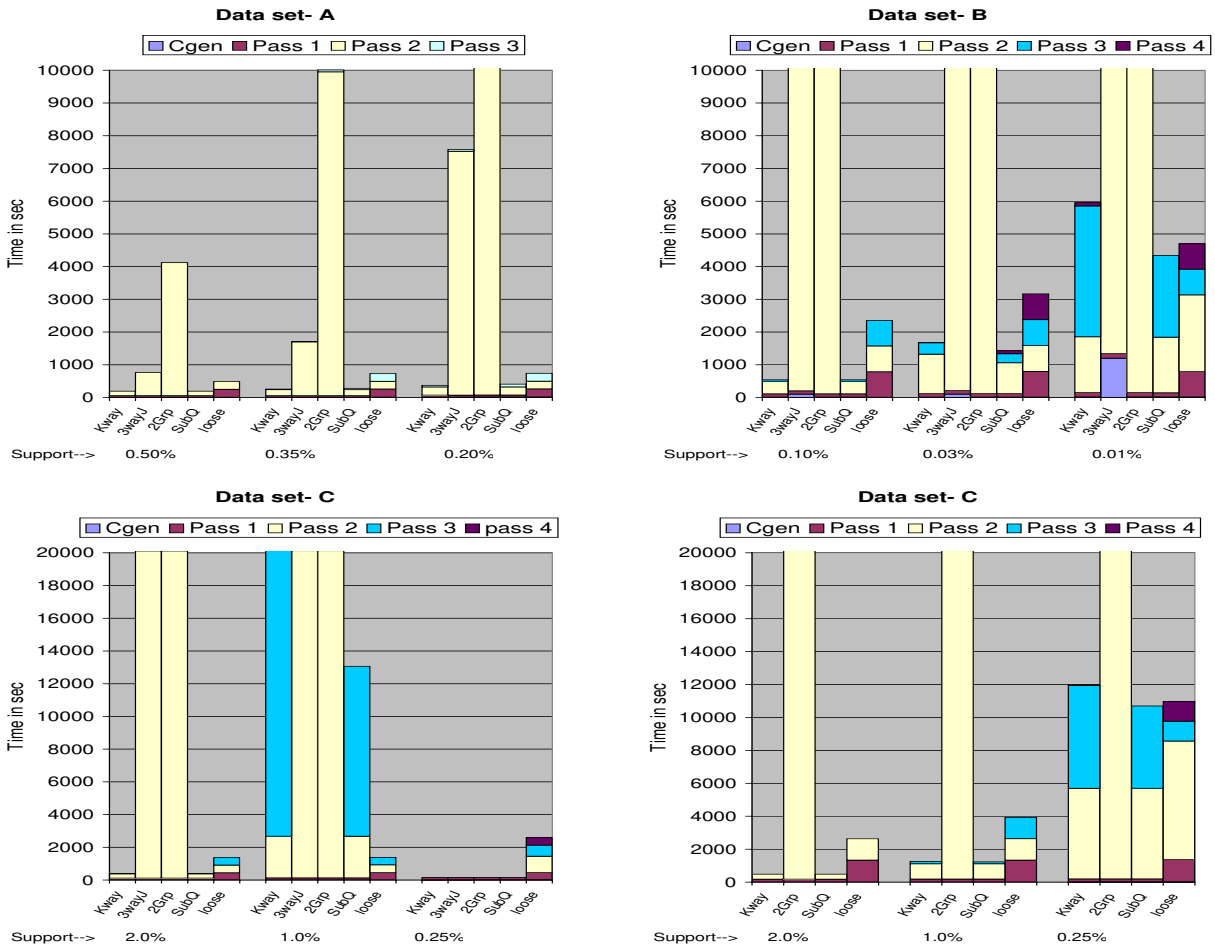


Figure Comparison of four SQL approaches The time taken is broken down by each pass and the candidate generation time

itemsets Although the Subquery approach is comparable or better than the Loose coupling approach in some cases for other cases involving low support values it did not complete even after taking ten times more time than the Loose coupling coupling approach For instance for Dataset C for the high support value of the Subquery approach is better than the Loose coupling approach But as we descrease the support to the Subquery approach becomes almost ten times worse than the Loose coupling coupling approach and on further decreasing the support to none of the SQL approaches could be taken to completion because of storage over ow

The GroupBy approach is signi cantly worse than the other two approaches because it involves an index ORing operation on k indices for each pass k of the algorithm In addition the inner group by requires sorting a large intermediate relation The outer group by is comparatively faster because the sorted result is of size at most C which is much smaller than the result size of the inner group by The DBMS does aggregation during sorting therefore the size of the result is an important factor in the total cost

The wayJoin approach is worse than the KwayJoin approach because it cannot use the pass optimization of KwayJoin and it requires writing large intermediate relatons As shown in AMS there might be other datasets especially ones where there is signi cant reduction in the size of T as k increases where wayJoin might perform better than KwayJoin One disadvantage of the wayJoin approach is that it requires space to store and log the temporary relations T generated in each pass

The important conclusion we drew from this study therefore is that implementations based on pure SQL are too slow to be considered a general alternative to the existing Loose coupling approach

Support counting using SQL with object relational extensions

In this section we study approaches that use object relational features in SQL to improve performance We rst consider an approach we call GatherJoin and its three variants in Section Next we present a very di erent approach called Vertical in Section Finally in Section we present an approach based on SQL bodied functions For each approach we also outline a cost based analysis of the execution time to choose between these di erent approaches In Section we present performance comparisons

GatherJoin

The GatherJoin approach see Figure generates all possible k item combinations of items contained in a transaction joins them with the candidate table C and counts the support of the itemsets by grouping the join result It uses two table functions Gather and Comb K The data table T is scanned in the tid item order and passed to the table function Gather which collects all the items of a transaction in memory and outputs a record for each transaction Each record consists of two attributes the tid and item list which is a collection of all items in a eld of type VARCHAR or BLOB The output of Gather is passed to another table function Comb K which returns all k item combinations formed out of the items of a transaction A record output by Comb K has k attributes T_itm T_itm which can be used to probe into the C table An index is constructed on all the items of C to make the probe e cient

This approach is analogous to the KwayJoin approach except that we have replaced the k way self join of T with the table functions Gather and Comb K These table functions are easy to code and do not require a large amount of memory It is also possible to merge them into a single table function GatherComb K which is what we did in our implementation Note that the Gather function is not required when the data is already in a horizontal format where each tid is followed by a collection of all its items

```
insert into F   select item        item    count
from C
        select t  T_itm        t  T_itm   from T
       table  Gather T tid  T item   as t
       table  Comb K t  tid  t  item list   as t
where t  T_itm      C  item   and


       t  T_itm      C  item
group by C  item        C  item
having count         minsup
```

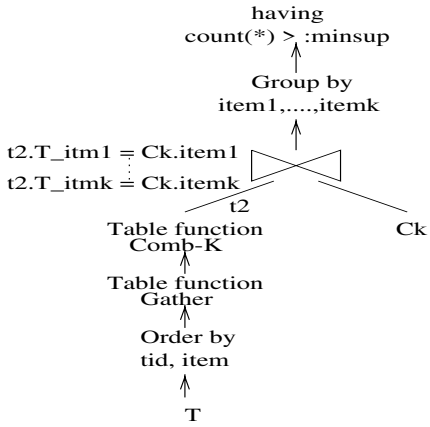


Figure Support Counting by

Special pass optimization For k the candidate set C is simply a join of F with itself Therefore we can optimize the pass by replacing the join with C by a join with F before the table function see Figure The table function now gets only frequent items and generates signi cantly fewer item combinations We apply this optimization to other passes too However unlike pass we still have to do the nal join with C and therefore the bene t is not as signi cant

```
insert into F   select tt  T_itm    tt  T_itm    count
from  select   from T  F  where
      T item    F  item   as tt
      table  GatherComb   tid item   as tt
group by tt  T_itm    tt  T_itm
having count         minsup
```

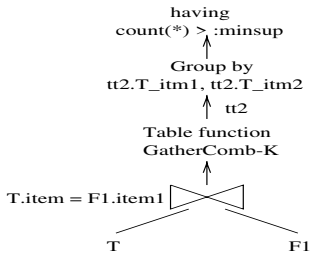


Figure Support Counting by in the second pass

Variations of GatherJoin approach

GatherCount One variation of the GatherJoin approach for pass two is the GatherCount approach where we perform the group by inside the table function GatherComb We will refer to this extended table function as Gather Cnt The candidate itemset C is represented as a two dimensional array as suggested in AMS inside function Gather Cnt Instead of outputting the item combinations the function uses the combinations

to directly update support counts in memory and outputs only the frequent itemsets F and their support after the last transaction

The attraction of this otion is the absence of the outer grouping The UDF code is small since it only needs to maintain a D array We could apply the same trick for subsequent passes but the coding becomes considerably more complicated because of the need to maintain hash tables to index the C s The disadvantage of this approach is that it can require a large amount of memory to store C If enough memory is not available C needs to be partitioned and the process has to be repeated for each partition Another problem with this approach is that it cannot be automatically parallelized

GatherPrune A problem with the GatherJoin approach is the high cost of joining the large number of item combinations with C We can push the join with C inside the table function and thus reduce the number of such combinations C is converted to a BLOB and passed as an argument to the table function

The cost of passing the BLOB for every tuple of R can be high In general we can reduce the parameter passing cost by using a smaller Blob that only approximates the real C The trade o is increased cost for other parts notably grouping because not as many combinations are ltered A problem with this approach is the increased coding complexity of the table function

Horizontal This is another variation of GatherJoin that rst uses the Gather function to transform the data to the horizontal format but is otherwise similar to the GatherJoin approach Rajamani et al RIC propose nding associations using a similar approach augmented with some pruning based on a variation of the GatherPrune approach Their results assume that the data is already in a horizontal format which is often not true in practice They report that their SQL implementation is two to six times slower than a UDF implementation

R	number of records in the input transaction table
T	number of transactions
N	average number of items per transaction –
F	number of frequent items
S C	sum of support of each itemset in set C
R	number of records out of R involving frequent items S F
N	average number of frequent items per transaction —
C	number of candidate k itemsets
C N k	number of combinations of size k possible out of a set of size n ———
s	cost of generating a k item combination using table function Comb k
group n m	cost of grouping n records out of which m are distinct
join n m r	cost of joining two relations of size n and m to get a result of size r
blob n	cost of passing a BLOB of size n integers as an argument

Table Notations used for cost analysis of di erent approaches

Cost analysis of GatherJoin and its variants The relative performance of these variants depends on a number of data characteristics like the number of items total number of transactions average length of a transaction etc We express the costs in each pass in terms of parameters that are known or can be estimated after the candidate generation step of each pass The purpose of this analysis is to help us choose between the di erent options Therefore instead of including all I O and CPU costs we include only those terms that help us distinguish between di erent options We use the notations of Table in the cost analysis

The cost of GatherJoin includes the cost of generating k item combinations joining with C and grouping to count the support The number of k item combinations generated T is C N k T Join with C lters out the non candidate item combinations The size of the join result is the sum of the support of all the candidates denoted by S C The actual value of the support of a candidate itemset will be known only after the support counting phase However we approximate it to the minimum of the support of all its k subsets in F The total cost of the GatherJoin approach is

T s join T C S C group S C C where T C N k T

The above cost formula needs to be modi ed to re ect the special optimization of joining with F to consider only frequent items We need a new term join R F R and need to change the formula for T to include only frequent items N instead of N

For the second pass we do not need the outer join with C The total cost of GatherJoin in the second pass is

$$\text{join } R\ F\ R \quad T \quad s \quad \text{group } T\ C \quad \text{where } T \quad C\ N \quad T \ \frac{N \quad T}{}$$

Cost of GatherCount in the second pass is similar to that for basic GatherJoin except for the nal grouping cost

join R F R group_internal T C F s

In this formula group_internal denotes the cost of doing the support counting inside the table function

For GatherPrune the cost equation is

R blob k C S C s group S C C

We use blob k C for the BLOB passing cost since each itemset in C contains k items

The cost estimate of Horizontal is similar to that of GatherJoin except that here the data is materialized in the horizontal format before generating the item combinations

Vertical

We rst transform the data table into a vertical format by creating for each item a BLOB containing all tids that contain that item Tid list creation phase and then count the support of itemsets by merging together these tid lists support counting phase This approach is similar to the approaches in SON ZPOL For creating the Tid lists we use a table function Gather This is the same as the Gather function in GatherJoin except that we create the tid list for each frequent item The data table T is scanned in the item tid order and passed to the function Gather The function collects the tids of all tuples of T with the same item in

memory and outputs a item tid list tuple for items that meet the minimum support criterion The tid lists are represented as BLOBs and stored in a new TidTable with attributes item tid list

In the support counting phase for each itemset in C we want to collect the tid lists of all k items and use a UDF to count the number of tids in the intersection of these k lists The tids are in the same sorted order in all the tid lists and therefore the intersection can be done e ciently by a single pass of the k lists This step can be improved by decomposing the intersect operation to share these operations across itemsets having common pre xes as follows

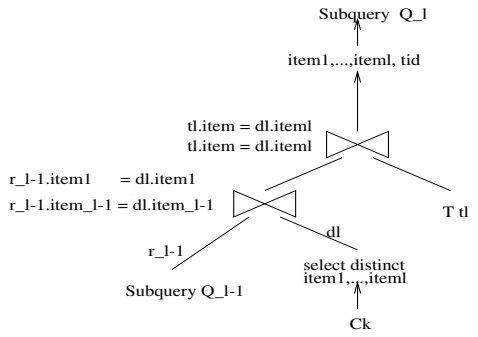
We rst select distinct item item pairs from C For each distinct pair we rst perform the intersect operation to get a new result tidlist then nd distinct triples item item item from C with the same rst two items intersect result tidlist with tid list for item for each triple and continue with item and so on until all k tid lists per itemset are intersected This approach is analogous to the Subquery approach presented for SQL

The above sequence of operations can be written as a single SQL query for any k as shown in Figure The nal intersect operation can be merged with the count operation to return a count instead of the tid list w e do not show this optimization in the query of Figure for simplicity

```
insert into F select item item count tid list as cnt
    from Subquery Q t where cnt minsup

Subquery Q for any l between and k
    select item item
      Intersect r tid list t tid list as tid list
    from TidTable t Subquery Q as r
      select distinct item item from C as d
    where r item d item and and
        r item d item and
        t item d item

Subquery Q select from TidTable
```



Tree diagram for Subquery

Figure Support counting using UDF

Special pass optimization For pass we need not generate C and join the TidTables with C Instead we perform a self join on the TidTable using predicate t item t item

```
insert into F select t item t item cnt
from select item item CountIntersect t tid list t tid list as cnt
        from TidTable t TidTable t
        where t item t item as t
where cnt minsup
```

Cost analysis The cost of the Vertical approach during support counting is dominated by the cost of invoking the UDFs and intersecting the tid lists The UDF is rst called for each distinct item pair in C then for each distinct item triple and so on Let d be the number of distinct j item tuples in C Then the number of UDF invocations is d In each invocation two BLOBs of tid list are passed as arguments The UDF intersects the tid lists by a merge pass and hence the cost is proportional to average length of a tid list The average length of a tid list can be approximated to — Note that with each intersect the tid list keeps shrinking However we ignore such e ects for simplicity

The total cost of the Vertical approach is

$$d \qquad \text{Blob}\frac{R}{F} \qquad \text{Intersect}\frac{R}{F}$$

In the above formula Intersect n denotes the cost of intersecting two tid lists with a combined size of n We are not including the join costs in this analysis because it accounted for only a small fraction of the total cost

The total cost of the second pass is

$$C \qquad \text{Blob}\frac{R}{F} \qquad \text{Intersect}\frac{R}{F}$$

SQL bodied functions SBF

This approach is based on SQL bodied procedures commonly known as SQL PSM MM SQL PSMs extend SQL with additional control structures We will make use of one such construct for do end

We use the for construct to scan the transaction table T in the tid item order Then for each tuple tid item of T we update those tuples of C that contain one matching item C is extended with extra attributes prevTid match supp The prevTid attribute keeps the tid of the previous tuple of T that matched that itemset The match attribute contains the number of items of prevTid matched so far and supp holds the current support of that itemset

```
for this as select    from T do
    update C   set prevTid     tid
               match     case when tid     prevTid then match     else    end
               supp     case when match     k    and tid     prevTid then supp    else supp end
    where item     item   or


              item     item
end for
insert into F   select item          item    supp
    from C   where supp      minsupp
```

We omit a cost analysis of this approach because it was not competitive compared to other approaches

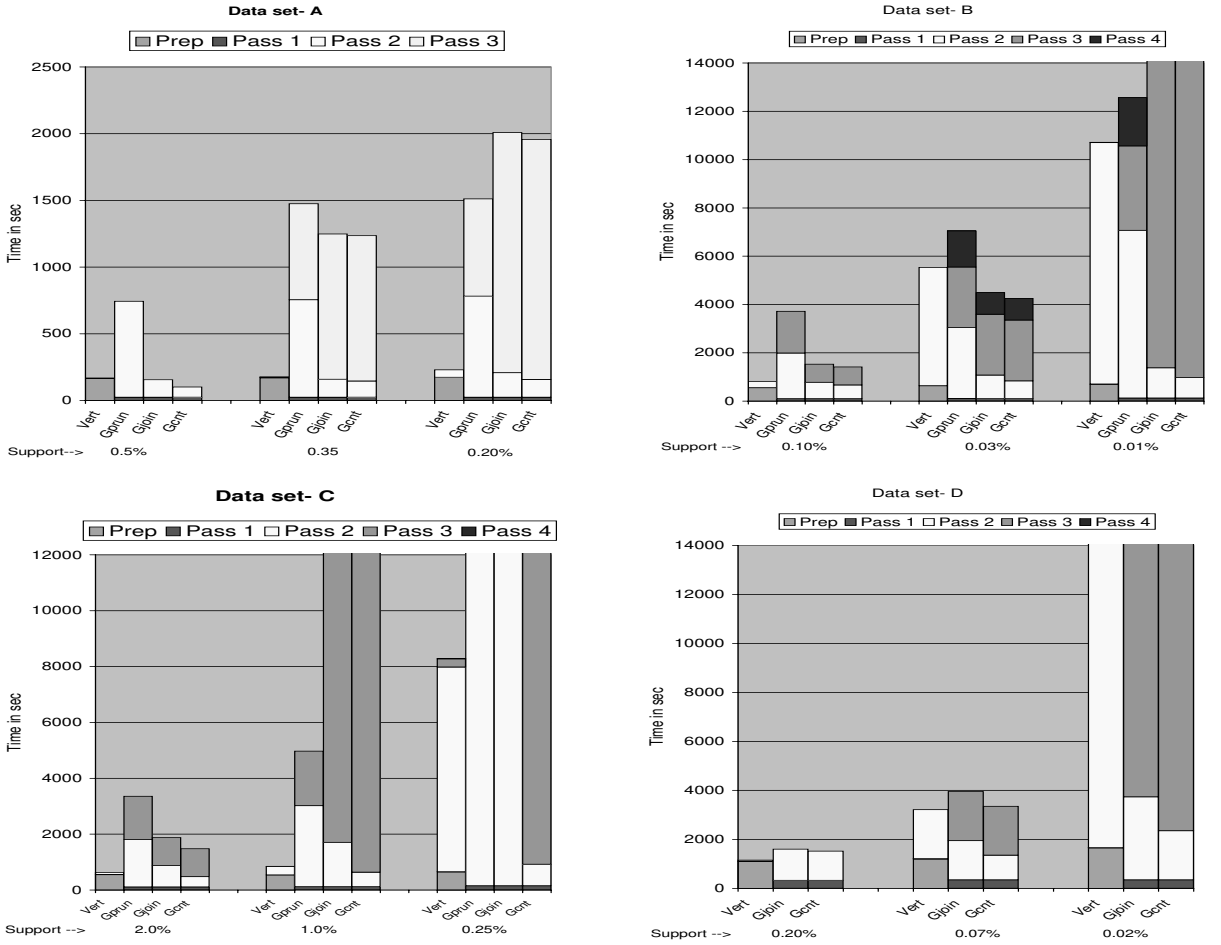


Figure Comparison of four SQL OR approaches and on four datasets for di erent support values The time taken is broken down by each pass and an initial prep stage where any one time data transformation cost is included

Performance comparison of SQL OR approaches

We studied the performance of six SQL OR approaches using the datasets summarized in Table Figure shows the results for only four approaches GatherJoin GatherCount GatherPrune and Vertical For the other two approaches Horizontal and SBF the running times were so large that we had to abort the runs in many cases The reason why the Horizontal approach was signi cantly worse than the GatherJoin approach was the time to transform the data to the horizontal format For instance for Dataset C it was hours which is almost times more than the time taken by Vertical for support For Dataset B the process was aborted after running for hours After the transformation compared to GatherJoin the time taken by Horizontal was also signi cantly worse when run without the frequent itemset ltering optimization but with the optimization the performance was comparable The SBF approach had signi cantly worse performance because of the expensive indexing ORing of the k join predicates Another problem with this approach is the large number of updates to the C table In DB all of these updates are logged resulting in severe performance degradation

We rst concentrate on the overall comparison between the di erent approaches Then we will compare the approaches based on how they perform in each pass of the algorithm

The Vertical approach has the best overall performance and it is sometimes more than an order of magnitude better than the other three approaches

The majority of the time of the Vertical approach is spent in transforming the data to the Vertical format in most cases shown as prep in gure The vertical representation is like an index on the item attribute If we think of this time as a one time activity like index building then performance looks even better The time to transform the data to the Vertical format was much smaller than the time for the horizontal format although both formats write almost the same amount of data The reason is the di erence in the number of records written The number of frequent items is often two to three orders of magnitude smaller than the number of transactions

Between GatherJoin and GatherPrune neither strictly dominates the other The special pass optimization in GatherJoin had a big impact on performance With this optimization for Dataset B with support the running time for pass was reduced from hours to minutes

When we compare these approaches based on time spent in each pass no single approach emerges as the best for all passes of the with datasets

For pass three onwards Vertical is often two or more orders of magnitude better than the other approaches Even in cases like Dataset B support where it spends three hours in the second pass the total time for next two passes is only seconds whereas it is more than an hour for the other two approaches For higher passes the performance degrades dramatically for GatherJoin because the table function GatherCombK generates a large number of combinations For instance for pass of Dataset C even for support value of pass did not complete after hours whereas for Vertical pass nished in seconds GatherPrune is better than GatherJoin for third and later passes For pass GatherPrune is worse because the overhead of passing a large object as an argument dominates cost

The Vertical approach sometimes spends too much time in the second pass In some of these cases the GatherJoin approach was better in the second pass for instance for low support values of Dataset B whereas in other cases for instance Dataset C with minimum support GatherCount was the only good option In the latter case both GatherPrune and GatherJoin did not complete after more than six hours for pass Further they caused a storage over ow error because of the large size of the intermediate results to be sorted We had to divide the dataset into four equal parts and ran the second pass independently on each partition to avoid this problem

Two factors that a ect the choice amongst the Vertical GatherJoin and GatherCount approaches in di erent passes and pass in particular are number of frequent items F and the average number of frequent items per transaction N From Figure we notice that as the value of the support is decreased for each dataset causing the size of F to increase the performance of pass of the Vertical approach degrades rapidly This trend is also clear from our cost formulae The cost of the Vertical approach increases quadratically with F GatherJoin depends more critically on the number of frequent items per transaction For Dataset B even when the size of F increases by a factor of the value of N remains close to therefore the time taken by GatherJoin does not increase as much However for Dataset C the size of N increases from to as the support is decreased from to causing GatherJoin to deteriorate rapidly From the cost formula for

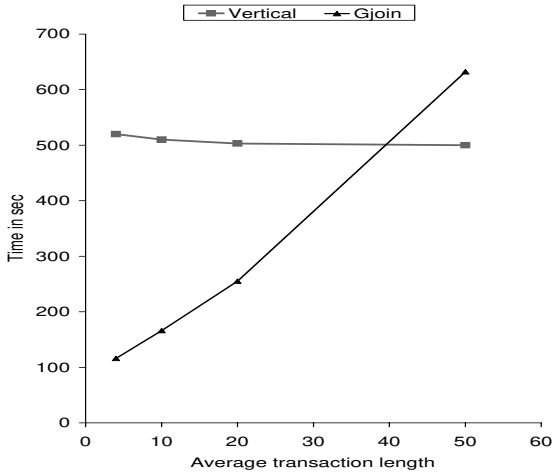


Figure E ect of increasing transaction length average number of items per transaction

GatherJoin we notice that the total time for pass increases almost quadratically with N

We validated this observation further by running experiments on synthetic datasets for varying values of the number of frequent items per transaction We used the synthetic dataset generator described in AMS for this purpose We varied the transaction length the number of transactions and the support values while keeping the total number of records and the number of frequent items xed In Figure we show the total time spent in pass of the Vertical and GatherJoin approaches As the number of items per transaction transaction length increases the cost of Vertical remains almost unchanged whereas the cost of GatherJoin increases

Final hybrid approach

The previous performance section helps us draw the following conclusions Overall the Vertical approach is the best option especially for higher passes When the size of the candidate itemsets is too large the performance of the Vertical approach could su er In such cases GatherJoin is a good option as long as the number of frequent items per transaction N is not too large When N is large GatherCount may be the only good option even though it may not easily parallelize

The hybrid scheme chooses the best of the three approaches GatherJoin GatherCount and Vertical for each pass based on the cost estimates outlined in Sections and The parameter values used for the estimation are available at the end of the previous pass In Section we plot the nal running time for the di erent datasets based on this hybrid approach

Architecture comparisons

In this section our goal is to compare the ve alternatives Loose coupling Stored procedure Cache Mine UDF and the best SQL implementation

For Loose coupling we use the implementation of the Apriori algorithm AMS for nding association rules provided with the IBM data mining product Intelligent Miner Int For Stored procedure we extracted

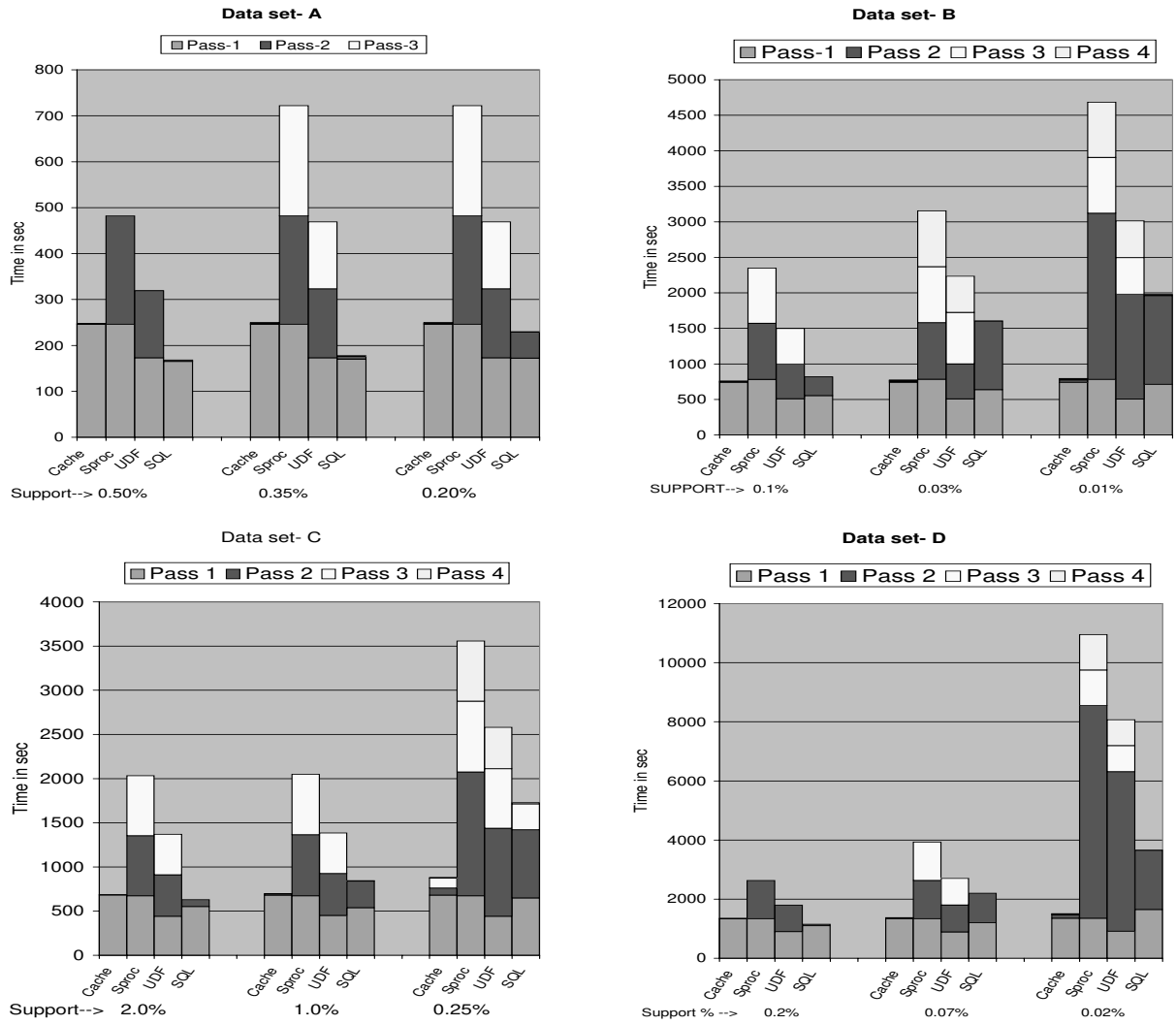


Figure Comparison of four architectures and on four real life datasets is similar to For each dataset three di erent support values are used The total time is broken down by the time spent in each pass

the Apriori implementation in Intelligent Miner and created a stored procedure out of it The stored procedure is run in the unfenced mode in the database address space For Cache Mine we used an option provided in Intelligent Miner that causes the input data to be cached as a binary le after the rst scan of the data from the DBMS The data is copied in the horizontal format where each tid is followed by an encoding of all its frequent items For the UDF architecture we use the UDF implementation of the Apriori algorithm described in AS b In this implementation rst a UDF is used to initialize state and allocate memory for candidate itemsets Next for each pass a collection of UDFs are used for generating candidates counting support and checking for termination These UDFs access the initially allocated memory address of which is passed around in BLOBs Candidate generation creates the in memory hash trees of candidates This happens entirely in the UDF without any involvement of the DBMS During support counting the data table is scanned sequentially and for each tuple a UDF is used for updating the counts on the memory resident hashtree

Timing comparison

In Figure we show the performance of Cache Mine Stored procedure UDF and the hybrid SQL OR implementation for the datasets in Table We do not show the times for the Loose coupling option because its performance was very close to the Stored procedure option

We can make the following observations

Cache Mine has the best or close to the best performance in all cases of its total time is spent in the rst pass where data is accessed from the DBMS and cached in the le system Compared to the SQL approach this approach is a factor of to times faster

The Stored procedure approach is the worst The di erence between Cache Mine and Stored procedure is directly related to the number of passes For instance for Dataset A the number of passes increases from two to three when decreasing support from to causing the time taken to increase from two to three times The time spent in each pass for Stored procedure is the same except when the algorithm makes multiple passes over the data since all candidates could not t in memory together This happens for the lowest support values of Dataset B Dataset C and Dataset D Time taken by Stored procedure can be expressed approximately as number of passes times time taken by Cache Mine

UDF is similar to Stored procedure The only di erence is that the time per pass decreases by for UDF because of closer coupling with the database

The SQL approach comes second in performance after the Cache Mine approach for low support values and is even somewhat better for high support values The cost of converting the data to the vertical format for SQL is typically lower than the cost of transforming data to binary format outside the DBMS for Cache Mine However after the initial transformation subsequent passes take negligible time for Cache Mine For the second pass SQL takes signi cantly more time than Cache Mine particularly when we decrease support For subsequent passes even the SQL approach does not spend too much time Therefore the di erence between Cache Mine and SQL is not very sensitive to the number of passes because both approaches spend negligible time in higher passes

The SQL approach is to times better than Stored procedure or Loose coupling approach As we decreased the support value so that the number of passes over the dataset increases the gap widens

Note that we could have implemented Stored procedure using the same hybrid algorithm that we used for SQL instead of using the IM algorithm Then we expect the performance of Stored procedure to improve because the number of passes to the data will decrease However we will pay the storage penalty of making additional copy of the data as we did in the Cache Mine approach The performance of Stored procedure cannot be better than Cache Mine because as we have observed that most of the time of Cache Mine is spent in the rst pass which cannot be changed for Stored procedure

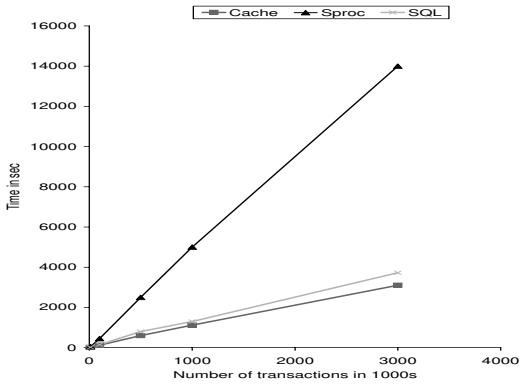


Figure Scale up with increasing number of transac tions

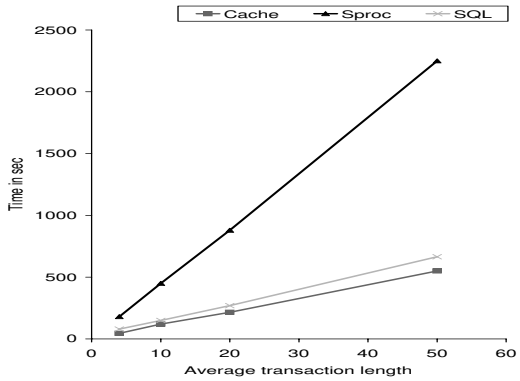


Figure Scale up with increasing transaction length

Scale up experiment Our experiments with the four real life datasets above has shown the scaling property of the di erent approaches with decreasing support value and increasing number of frequent itemsets We experiment with synthetic datasets to study other forms of scaling increasing number of transactions and increasing average length of transactions Figure shows how Stored procedure Cache Mine and SQL scale with increasing number of transactions UDF and Loose coupling have similar scale up behavior as Stored procedure therefore we do not show these approaches in the gure We used a dataset with average number of items per transaction thousand total items and a default pattern length de ned in AMS of Thus the size of the dataset is times the number of transactions As the number of transactions is increased from K to K the time taken increases proportionately The largest frequent itemset was long This explains the ve fold di erence in performance between the Stored procedure and the Cache Mine approach Figure shows the scaling when the transaction length changes from to while keeping the number of transactions xed at K All three approaches scale linearly with increasing transaction length

Impact of longer names In these experiments we assumed that the tids and item ids are all integers Often in practice these are character strings longer than four characters Longer strings need more storage and cost more during comparisons This could hurt all four of the alternatives For the Stored procedure UDF and Cache Mine approach the time taken to transfer data will increase The Intelligent Miner code Int maps all character elds to integers using an in memory hash table Therefore beyond the increase in the data transfer and mapping costs which accounts for the bulk of the time we do not expect the processing time of these three alternatives to increase For the SQL approach we cannot assume an in memory hash table for doing the mapping therefore we use an alternative approach based on table functions

For SQL approach we discuss the hybrid approach The two already expensive steps that could su er because of longer names are nal group bys during pass or higher when the GatherJoin approach is chosen and tid list operations when the Vertical approach is chosen For e cient performance the rst step requires a mapping of item ids and the second one requires us to map tids We use a table function to map the tids to unique integers e ciently in one pass and without making extra copies The input to the table function is the data table in the tid order The table function remembers the previous tid and the maintains a counter Every time the tid changes the counter is incremented This counter value is the mapping assigned to each tid We need to do the tid mapping only once before creating the TidTable in the Vertical approach and therefore we can pipeline these two steps The item mapping is done slightly di erently After the rst pass we add a column to F containing a unique integer for each item We do the same for the TidTable The GatherJoin approach already joins the data table T with F before passing to table function Gather Therefore we can pass to Gather the integer mappings of each item from F instead of its original character representation After these two transformations the tid and item elds are integers for all the remaining queries including candidate generation and rule generation By mapping the elds this way we expect longer names to have similar performance impact on all of our architectural options

Space overhead of di erent approaches

In Figure we summarize the space required for three options Stored procedure Cache Mine and SQL We assume that the tids and items are integers The space requirements for UDF and Loose coupling is the same as that for Stored procedure which in turn is less than the space needed by the Cache Mine and SQL approaches The Cache Mine and SQL approaches have comparable storage overheads For Stored procedure and UDF we do not need any extra storage for caching However all three options Cache Mine Stored procedure and UDF require data in each pass to be grouped on the tid In a relational DBMS we cannot assume any order on the physical layout of a table unlike in a le system Therefore we need either an index on the data table or need to sort the table every time to ensure a particular order Let R denote the total number of tid item pairs in the data table Either option has a space overhead of R integers The Cache Mine approach caches the data in an alternative binary format where each tid is followed by all the items it contains Thus the size of the cached data in Cache Mine is at most R T integers where T is the number of transactions For SQL we use the hybrid Vertical option This requires creation of an initial TidTable of size at most I R where I is the number of items Note that this is slightly less than the cache required by the Cache Mine approach The SQL approach needs to sort data in pass in all cases and pass in some cases where we used the GatherJoin approach instead of the Vertical approach This explains the large space requirement for Dataset B

In summary the UDF and Stored procedure approaches require the least amount of space followed by the Cache Mine and the SQL approaches which require roughly as much extra storage as the data When the item ids or tids are character strings instead of integers the extra space needed by Cache Mine and SQL is a much smaller fraction of the total data size because before caching we always convert item ids to their compact integer representation and store in binary format

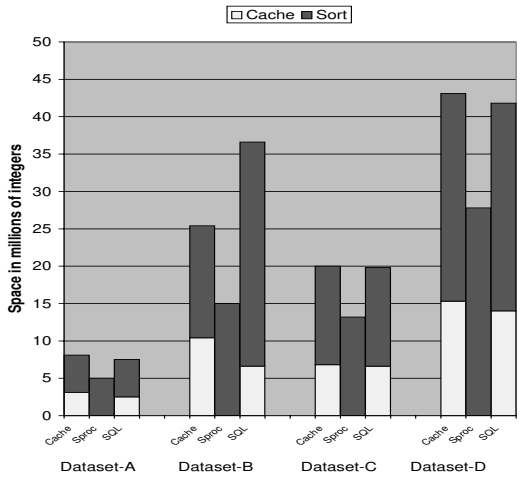


Figure Comparison of di erent architectures on space requirements The rst part refers to the space used in caching data and the second part refers to any temporary space used by the DBMS for sorting or alternately for constructing indices to be used during sorting The size of the data is the same as the space utilization of the approach

Summary of comparison between di erent architectures

We present a summary of the pros and cons of the di erent architectures on each of the following yard sticks a performance execution time b storage overhead c potential for automatic parallelization d development and maintenance ease e portability f inter operability

In terms of performance the Cache Mine approach is the best option The SQL approach is a close second it is always within a factor of two of Cache Mine for all of our experiments and is sometimes even slightly better The UDF approach is better than the Stored procedure approach by to Between Stored procedure and Cache Mine the performance di erence is a function of the number of passes made on the data if we make four passes of the data the Stored procedure approach is four times slower than Cache Mine Some of the recent proposals Toi BMUT that attempt to minimize the number of data passes to or might be useful in reducing the gap between the Cache Mine and the Stored procedure approach

In terms of space requirements the Cache Mine and the SQL approach loose to the UDF or the Stored procedure approach The Cache Mine and SQL approaches have similar storage requirements

The SQL implementation has the potential for automatic parallelization particularly on a SMP machine Parallelization could come for free for SQL queries Unfortunately the SQL option is too slow to be a candidate for parallelization The stumbling block for automatic parallelization using SQL OR could be queries involving UDFs that use scratch pads The only such function in our queries is the Gather table function This function essentially implements a user de ned aggregate and would have been easy to parallelize if the DBMS provided support for user de ned aggregates or allowed explicit control from the application about how to partition the data amongst di erent parallel instances of the function On a MPP machine although one could rely on the DBMS to come up with a data partitioning strategy it might be possible to better tune

performance if the application could provide hints about the best partitioning AS a Further experiments are required to assess how the performance of these automatic parallelizations would compare with algorithm speci c parallelizations e g AS a

The development time and code size using SQL could be shorter if one can get e cient implementations out of expressing the mining algorithms declaratively using a few SQL statements Thus one can avoid writing and debugging code for memory management indexing and space management all of which are already provided in a database system Note that these same code reuse advantages can be obtained from a well planned library of mining building blocks However there are some detractors to easy development using the SQL alternative First any attached UDF code will be harder to debug than stand alone C code due to lack of debugging tools Second stand alone code can be debugged and tested faster when run against at le data Running against at les is typically a factor of ve to ten faster compared to running against data stored in DBMS tables Finally some mining algorithms e g neural net based might be too awkward to express in SQL

The ease of porting of the SQL alternative depends on the kind of SQL used Within the same DBMS porting from one OS platform to another requires porting only the small UDF code and hence is easy In contrast the Stored procedure and Cache Mine alternatives require porting larger lines of code Porting from one DBMS to another could get hard for SQL approach if non standard DBMS speci c features are used For instance our preferred SQL implementation relies on the availability of DB s table functions for which the interface is still not standardized across other major DBMS vendors Also if di erent features have di erent performance characteristics on di erent database systems considerable tuning would be required In contrast the Stored procedure and Cache Mine approach are not tied to any DBMS speci c features The UDF implementation has the worst of both worlds it is large and is tied to a DBMS

One attraction of SQL implementation is inter operability and usage exibility The adhoc querying support provided by the DBMS enables exible usage and exposes potential for pipelining the input and output operators of the mining process with other operators in the DBMS However to exploit this feature one needs to implement the mining operators inside the DBMS This would require major rework in existing database systems The SQL approach presented here is based on embedded SQL and as such cannot provide operator pipelining and inter operability Queries on the mined result is possible with all four alternatives as long as the mined results are stored back in the DBMS

Other associations algorithms There are recent proposals Toi BMUT that attempt to min imize the number of data passes in the Apriori algorithm to two or three Currently the Stored procedure approach makes as many passes as the length of the largest itemset and in each pass most of the time is spent in reading the data from the DBMS Therefore by using these alternative algorithms it is possible to reduce the time spent by Stored procedure by as many times as the number of passes However the Cache Mine approach is almost a lower bound on the best performance possible for doing associations outside the DBMS because it spends most of its time in the rst pass which is the minimum amount of work any mining algorithm that works outside the DBMS has to spend read all of the data at least once Since the SQL approach is competitive with the one pass Cache Mine approach we expect our conclusions to hold for other algorithms that spend to times the time spent by Cache Mine These alternative algorithms are not likely to help improve the performance of the SQL implementation because they are aimed at reducing the number of passes over the data We have seen that with the SQL implementations hardly any time is spent on passes beyond because of the

Vertical approach The time spent in pass is not reduced because they all require counting support of all of C on the entire data anyway

Proposed extensions

One of the goals of our work has been to unbundle complex mining operations like associations and classi cations into smaller primitives that can be supported e ciently by a general purpose DBMS and be useful for a large class of mining algorithms Our exercise with association rules has helped us identify a few such primitives that we believe could be generally useful in a number of other decision support applications

Richer set operations We expect a richer collection of set operations to be useful for mining For associa tion rules we used three di erent versions of the basic subset operation For candidate generation we needed to nd all k subsets of a set of k elements for doing the pruning We enumerated the subsets explicitly in the query predicate because the size of the set and the subsets was xed and known in advance For support counting we used the Comb K table function which generated all k subsets of a variable length set For rule generation we used the GenSubset table function to generate all subsets of a variable length set The subset operation would also be useful in building decision trees on categorical attributes MAR

Another important set operation was the intersect operation used in the Vertical approach Most systems already have internal implementations of this operation for doing AND and OR operations on RID record identi er lists obtained during multiple index scans MHWC In current OLAP and data warehousing systems this operation is rampant in the popular bit mapped indices Our Vertical approach is curiously similar to this bit mapped approach with one important di erence Instead of performing ANDs on RIDs we perform the ANDs on another attribute the transaction identi er

Gather scatter operations Another common operation is the Gather operation that can transform two attributes in a data table to a form where for distinct values of one of the attributes called the grouping attribute we collect together in a set all values of the other attribute We can think of Gather as a glori ed aggregate function Its reverse operation scatter is a special case of the subset operation where the subset size is

We used this operation in four of our SQL approaches GatherJoin GatherPrune Vertical and Horizontal To implement the Gather table function in our queries we required data to appear in a particular order as inputs to the table function This would be trivial to express in SQL if order by clauses were allowed in inner subqueries In the absence of such a feature we relied on our knowledge of DB s internals to force such an order in our experiments Another way is to treat the Gather function as an aggregate function that simply concatenates its arguments Systems that have support for user de ned aggregate functions SK can easily support such a functionality

There are several other decision support applications where gather could be extremely useful For instance suppose we are trying to classify customers of a credit card company and the data in addition to static customer attributes like age salary consists of detailed transaction data about each purchase activity of a customer In this case we would like to gather all activities of a customer as one time series and extract features from these time series for classi cation In OLAP applications too data often needs to be converted back and forth from a

format where di erent measures are gathered together as di erent attributes of the same row to a format where di erent measures are scattered as di erent rows

Conclusion and future work

We explored various architectural alternatives for integrating mining with a relational database system As an initial step in that direction we studied the association rules algorithms with the twin goals of nding the trade o s between architectural options and the extensions needed in a DBMS to e ciently support mining We experimented with di erent ways of implementing the association rules mining algorithm in SQL to nd if it is at all possible to get competitive performance out of SQL implementations

We considered two categories of SQL implementations First we experimented with four di erent implementations based purely on SQL Experiments with real life datasets showed that it is not possible to get good performance out of pure SQL based approaches alone We next experimented with a collection of approaches that made use of the new object relational extensions like UDFs BLOBs Table functions etc With this extended SQL we got orders of magnitude improvement over the SQL based implementations

We compared the SQL implementation with di erent architectural alternatives We concluded that based just on performance the Cache Mine approach is the winner A close second is the SQL OR approach that was sometimes slightly better than Cache Mine and was never worse than a factor of two on our datasets Both these approaches require additional storage for caching however The Stored procedure approach does not require any extra space except possibly for initially sorting the data in the DBMS and can perhaps be made to be within a factor of two to three of Cache Mine with the recent algorithms Toi BMUT The UDF approach is a factor of to faster than Stored procedure but is signi cantly harder to code The SQL approach o ers some secondary advantages like easier development and maintenance and potential for automatic parallelization However it might not be as portable as the Cache Mine approach across di erent database management systems

This exercise has demonstrated that with a good understanding of the performance pro le of a DBMS engine it is possible to express the associations computation e ciently in SQL Often the SQL formulation may not be a one to one translation of the le system code therefore it might be necessary to design alternative methods This exercise also helped us identify some easy to implement primitives that are useful for other mining algorithms as well

The work presented in this paper points to several directions for future research A natural next step is to experiment with other kinds of mining operations e g clustering and classi cation FPSSU to verify if our conclusions about associations hold for these other cases too We experimented with generalized association rules SA and sequential patterns SA problems and found similar results In some ways associations is the easiest to integrate as the frequent itemsets can be viewed as generalized group bys Another useful direction is to explore what kind of a support is needed for answering short interactive adhoc queries involving a mix of mining and relational operations How much can we leverage from existing relational engines What data model and language extensions are needed Some of these questions are orthogonal to whether the bulky mining operations are implemented using SQL or not Nevertheless these are important in providing analysts with a well integrated platform where mining and relational operations can be inter mixed in exible ways

Acknowledgements We would like to acknowledge the help of Don Chamberlin Bobbie Cochrane Cli Leung Guy Lohman Eric Louie Hamid Pirahesh Bertold Reinwald Eugene Shekita Dave Simmens Amit Somani George Wilson and Swati Vora for providing help with DB and participating in discussions We would like to thank Ramakrishnan Srikant for helping us with the associations code and insightful feedback

References

AAB Rakesh Agrawal Andreas Arning Tony Bollinger Manish Mehta John Shafer and Ramakrishnan Srikant The Quest Data Mining System In Proc of the nd Int l Conference on Knowledge Discovery in Databases and Data Mining Portland Oregon August

AIS Rakesh Agrawal Tomasz Imielinski and Arun Swami Mining association rules between sets of items in large databases In Proc of the ACM SIGMOD Conference on Management of Data pages Washington D C May

AMS Rakesh Agrawal Heikki Mannila Ramakrishnan Srikant Hannu Toivonen and A Inkeri Verkamo Fast Discovery of Association Rules In Usama M Fayyad Gregory Piatetsky Shapiro Padhraic Smyth and Ramasamy Uthurusamy editors Advances in Knowledge Discovery and Data Mining chapter pages AAAI MIT Press

AS Rakesh Agrawal and Ramakrishnan Srikant Mining Sequential Patterns In Proc of the th Int l Conference on Data Engineering Taipei Taiwan March

AS a Rakesh Agrawal and John Shafer Parallel mining of association rules IEEE Transactions on Knowledge and Data Engineering December

AS b Rakesh Agrawal and Kyuseok Shim Developing tightly coupled data mining applications on a rela tional database system In Proc of the nd Int l Conference on Knowledge Discovery in Databases and Data Mining Portland Oregon August

BMUT Sergey Brin Rajeev Motwani Je rey D Ullman and Shalom Tsur Dynamic itemset counting and implication rules for market basket data In Proc of the ACM SIGMOD Conference on Management of Data May

Cha Don Chamberlin Using the New DB IBM s Object Relational Database System Morgan Kauf mann

FPSSU Usama M Fayyad Gregory Piatetsky Shapiro Padhraic Smyth and Ramasamy Uthurusamy edi tors Advances in Knowledge Discovery and Data Mining AAAI MIT Press

HFK J Han Y Fu K Koperski W Wang and O Zaiane DMQL A data mining query language for relational datbases In Proc of the SIGMOD workshop on research issues on data mining and knowledge discovery Montreal Canada May

HS Maurice Houtsma and Arun Swami Set oriented mining of association rules In Int l Conference on Data Engineering Taipei Taiwan March

IBM IBM Corporation DB Universal Database Application programming guide Version

IM T Imielinski and Heikki Mannila A database perspective on knowledge discovery Communication of the ACM Nov

Int Internationl Business Machines IBM Intelligent Miner User s Guide Version Release SH edition July

IVA T Imielinski A Virmani and A Abdulghani Discovery Board Application Programming Interface and Query Language for Database Mining In Proc of the nd Int l Conference on Knowledge Discovery and Data Mining Portland Oregon August

Kul Krishna Kulkarni Object oriented extensions in SQL a status report Sigmod record

MAR Manish Mehta Rakesh Agrawal and Jorma Rissanen SLIQ A fast scalable classi er for data mining In Proc of the Fifth Int l Conference on Extending Database Technology EDBT Avignon France March

MHWC C Mohan D Haderle Y Wang and J Cheng Single table access using multiple indexes op timization execution and concurrency control techniques In Proc International Conference on Extending Database Technology pages

MM Jim Melton and Nelson Mattos SQL a tutorial Twenty second international conference on Very large data bases tutorial September

MPC R Meo G Psaila and S Ceri A new SQL like operator for mining association rules In Proc of the nd Int l Conference on Very Large Databases Bombay India Sep

MS J Melton and A Simon Understanding the new SQL A complete guide Morgan Kau man

Ora Oracle Oracle RDBMS Database Administrator s Guide Volumes I II Version May

PR Hamid Pirahesh and Berthold Reinwald SQL table function open architecture and data access middleware In SIGMOD

RIC Karthick Rajamani Bala Iyer and Atul Chaddha Using DB s object relational extensions for mining associations rules Technical Report TR Santa Teresa Laboratory IBM Corporation sept

SA Ramakrishnan Srikant and Rakesh Agrawal Mining Generalized Association Rules In Proc of the st Int l Conference on Very Large Databases Zurich Switzerland September

SA Ramakrishnan Srikant and Rakesh Agrawal Mining Sequential Patterns Generalizations and Per formance Improvements In Proc of the Fifth Int l Conference on Extending Database Technology EDBT Avignon France March

SK M R Stonebraker and Greg Kemnitz The POSTGRES next generation database management system Communications of the ACM

SK A Siebes and M L Kersten KESO Minimizing Database Interaction In Proc of the rd Int l Conference on Knowledge Discovery and Data Mining Newport Beach California August

SON A Savasere E Omiecinski and S Navathe An e cient algorithm for mining association rules in large databases In Proc of the VLDB Conference Zurich Switzerland September

TAC Dick Tsur Serge Abiteboul Chris Clifton Rajeev Motwani and Svetlozar Nestorov Query ocks A generalization of association rule mining In SIGMOD to appear

Toi Hannu Toivonen Sampling large databases for association rules In Proc of the nd Int l Con ference on Very Large Databases pages Mumbai Bombay India September

ZPOL M J Zaki S Parthasarathy M Ogihara and W Li New Algorithms for Fast Discovery of Association Rules In Proc of the rd Int l Conference on Knowledge Discovery and Data Mining Newport Beach California August