Scalable Numerical Queries by Algebraic Inequality Transformations

Thanh Truong and Tore Risch

Department of Information Technology, Uppsala University
Box 337, SE-751 05, Sweden
thanh.truong@it.uu.se, tore.risch@it.uu.se

Abstract. To enable historical analyses of logged data streams by SQL queries, the Stream Log Analysis System (SLAS) bulk loads data streams derived from sensor readings into a relational database system. SQL queries over such log data often involve numerical conditions containing inequalities, e.g. to find suspected deviations from normal behavior based on some function over measured sensor values. However, such queries are often slow to execute, because the query optimizer is unable to utilize ordered indexed attributes inside numerical conditions. In order to speed up the queries they need to be reformulated to utilize available indexes. In SLAS the query transformation algorithm AQIT (Algebraic Query Inequality Transformation) automatically transforms SQL queries involving a class of algebraic inequalities into more scalable SQL queries utilizing ordered indexes. The experimental results show that the queries execute substantially faster by a commercial DBMS when AQIT has been applied to preprocess them.

1 Introduction

We first introduce a real-world scenario application under investigation in the Smart Vortex project [15], which requires queries involving numerical expressions. A factory operates some machines. On each machine, there are a number of sensors to measure different physical properties, e.g. power consumption, pressure, temperature, etc. The sensors generate logs of measurements per machine that carry a time stamp *ts*, a machine identifier *m*, a sensor identifier *s*, a measured value *mv*, and a measurement class *mc* for the kind of measurements made by the sensor. Examples of measurement classes are oil pressures of hydraulic filters and pressures of gear pumps. The logs are analyzed by bulk loading them into a relational DBMS. To speed up performance when analyzing sensors of the same kind on many different machines, there is one table for each *measurement class* of each kind of physical property. To avoid repetition of unchanged sensor readings, each measured value *mv* on machine *m* is associated with a valid time interval *bt* and *et* indicating the begin time and end time for *mv*, computed from the log time stamp *ts* when the data is bulk loaded. Hence, the measurement of class *mc=MC* on machines *m* will be stored in the table *measuresMC(m, s, bt, et, mv)*. These tables will contain large volumes of log data from many sensors of the same kind on different machines.

S.S. Bhowmick et al. (Eds.): DASFAA 2014, Part I, LNCS 8421, pp. 95–109, 2014.

After the data streams have been loaded into *measuresMC()*, the user can issue off-line historical queries to find errors on machines in the past by looking for abnormal values of *mv*. This often requires search conditions containing inequalities inside numerical expression. In our scenario, in order to improve the performance of inequality queries over *mv*, a B-tree index is added on each *measuresMC.mv*, denoted *idx(measuresMC.mv)*. The following are typical numerical query conditions on tables *measuresA*, and *measuresB* to identify faulty behaviors of machines:

- *C1:* Were the measurements of class A higher than a threshold v_0 = 15.6? We express the condition as $C1(mv): mv > v_0$.
- *C2:* Were the measurements of class A higher than $r_1 = 300$ above the expected value v_1 = 15.6? We express the condition as $C2(mv): mv - v_1 > r1$.
- *C3:* Were the measurements of class B outside the range $r_2 = 11$ from the ideal value $v_1 = 20$? We express the condition as $C3(mv): |mv - v_1| > r_2$.
- *C4:* Were the measurements of class B outside the range r_3 =20% from v_1 = 20? We express the condition as $C4(mv): \left|\frac{mv - v_1}{v_1}\right| > r_3$.

The above conditions can be expressed in SQL. Relational databases can handle SQL query conditions of type *C1* efficiently, since there is an ordered index *idx(measuresA.mv)*. However, in *C2-C4* the inequalities are not defined directly over the attribute *mv* but through some numerical expressions, which makes the query optimizer not utilizing the indexes and hence the queries will execute slowly. We say that the indexes *idx(measuresA.mv)* and *idx(measuresB.mv)* are *not exposed* in *C2-C4*. To speed up such queries, the DBMS vendors recommend that the user reformulates them [11] which often requires rather deep knowledge of low-level index operations.

To automatically transform a class of queries involving inequality expressions into more efficient queries where indexes are exposed, we have developed the query transformation algorithm *AQIT* (*Algebraic Query Inequality Transformation)*. We show that AQIT substantially improves performance for queries with conditions of type *C2-C4*, exemplified by analyzing logged abnormal behavior in our scenario. Without the proposed query transformations the DBMS will do a full scan, not utilizing any index.

AQIT transforms queries with inequality conditions on single indexed attributes to utilize range search operations over B-tree indexes. In general, AQIT can transform inequality conditions of form $F(mv)\ \psi\ \varepsilon$, where *mv* is a variable bound to an indexed attribute *A, F(mv)* is an expression consisting of a combination of *transformable* functions *T*, currently $T \in \{+, -, /, *, power, sqrt, abs\}$, and ψ is an inequality comparison $\psi \in \{\leq, \geq, <, >\}$. AQIT tries to reformulate inequality conditions into equivalent conditions, $mv\ \psi'\ F'(\varepsilon)$ that makes the index on attribute A, *idx(A)* exposed to the query optimizer. AQIT has a strategy to automatically determine ψ' and $F'(\varepsilon)$. If AQIT fails to transform the condition, the original query is retained. For example, AQIT is currently not applicable on multivariable inequalities, which are subjects for future work.

In summary, our contributions are:

1. We introduce the algebraic query transformation strategy AQIT on a class of numerical SQL queries. AQIT is transparent to the user and does not require manual reformulation of queries. We show that it substantially improves query performance.
2. The prototype system SLAS (Stream Log Analysis System) implements AQIT as a SQL pre-processor to a relational DBMS. Thus, it can be used on top of any relational DBMS. Using SLAS we have evaluated the performance improvements of AQIT on log data from industrial equipment in use.

This paper is organized as follows. Section 2 discusses related work. Section 3 presents some typical SQL queries where AQIT improves performance. Section 4 gives an overview of SLAS and its functionality. Section 5 presents the AQIT algebraic transformation algorithm on inequality expressions. Section 6 evaluates the scalability of applying AQIT for a set of benchmark queries based on the scenario database, along with a discussion of the results. Section 7 gives conclusions and follow-up future work.

2 Related Work

The recommended solution to utilize an index in SQL queries involving arithmetic expressions is to manually reformulate the queries so that index access paths are exposed to the optimizer [5] [11] [13]. However, it may be difficult for the database user to do such reformulations since it requires knowledge about indexing, the internal structure of execution plans, and how query optimization works. There are a number of tools [16] [11], which point out inefficient SQL statements but do not automatically rewrite them. In contrast, AQIT provides a transparent transformation strategy, which automatically transforms queries to expose indexes, when possible. If this is not possible, the query is kept intact.

Modern DBMSs such as Oracle, PostgreSQL, DB2, and SQL Server support *function indexes* [10] [8], which are indexes on the result of a complex function applied on row-attribute values. When an insertion or update happens, the DBMS computes the result of the function and stores the result in an index. The disadvantage of function indexes compared to the AQIT approach is that they are infeasible for ad hoc queries, since the function indexes have to be defined beforehand. In particular, function indexes are very expensive to build in a populated database, since the result of the expression must be computed for every row in the database. By contrast, AQIT does not require any pre-computations when data is loaded or inserted into the database. Therefore AQIT makes the database updates more efficient, and simplifies database maintenance.

Computer algebra systems like Mathematica [1] and Maple [4] and constraints database systems [7] [9] also transform inequalities. However, those systems do not have knowledge about database indexes as AQIT. The current implementation is a DBMS independent SQL pre-processor that provides the index specific query rewritings.

FunctionDB [2] also uses an algebraic query processor. However, the purpose of FunctionDB is to enable queries to continuous functions represented in databases, and it provides no facilities to expose database indexes.

Extensible indexing [6] aims at providing scalable query execution for new kinds of data by introducing new kinds of indexes. However, it is up to the user to reformulate the queries to utilize a new index. By contrast, our approach provides a general mechanism for utilizing indexes in algebraic expressions, which complements extensible indexing. In the paper we have shown how to expose B-tree indexes by algebraic rewrites. Other kinds of indexes would require other algebraic rules, which is a subject of future work.

3 Example Queries

A relational database that stores both meta-data and logged data from machines has the following three tables:

machine(m, mm) represents meta-data about each machine installation identified by *m* where *mm* identifies the machine model. There is a secondary B-tree index on *mm*.

sensor(m, s, mc, ev, ad, rd) stores meta-data about each sensor installation *s* on each machine *m*. To identify different kinds of measurements, e.g oil pressure, filter temperature etc., the sensors are classified by their *measurement class*, *mc*. Each sensor has some tolerance thresholds, which can be an absolute or relative error deviation, *ad* or *rd,* from the expected value *ev*. There are secondary B-tree indexes on *ev*, *ad,* and *rd*.

measuresMC(m, s, bt, et, mv) enables efficient analysis of the behavior of different kinds of measurements over many machine installations over time. The table stores measurements *mv* of class *MC* for sensor installations identified by machine *m* and sensor *s* in valid time interval *[bt,et)*. By storing *bt* and *et* temporal interval overlaps can be easily expressed in SQL [3][14]. There are B-tree indexes on *bt*, *et*, and *mv*.

We use the abnormality thresholds *@thA* for queries determining deviations in table *measuresA*, *@thB* for queries determining absolute deviation in table *measuresB,* and *@thRB* for queries determining relative deviation in table *measuresB*. We shall discuss these thresholds in Section 6 in greater details.

The following queries *Q1*, *Q2*, and *Q3* identify abnormalities:

- **Query Q1** finds when and on what machines, the pressure reading of class A was higher than *@thA* from its expected value:

```
1 SELECT va.m, va.bt, va.et
2 FROM   measures A va, sensor s
3 WHERE  va.m = s.m AND va.s = s.s AND va.mv > s.ev + @thA.
```

AQIT has no impact for query *Q1* since the index *idx(measuresA.mv)* is already exposed.

- **Query Q2** identifies abnormal behaviors based on absolute deviations: When and for what machines did the pressure reading of class B deviate more than *@thB* from its expected value? AQIT translates the query into the following SQL query *T2:*

```
  Q2:
1 SELECT vb.m, vb.bt, vb.et
2 FROM  measuresB vb, sensor s
3 WHERE vb.m =s.m AND vb.s=s.s
AND
4 abs(vb.mv - s.ev) > @thB
5
```

```
T2:
SELECT vb.m, vb.bt, vb.et
FROM   measuresB vb, sensor s
WHERE  vb.m=vb.m AND vb.s=s.s  AND
   ((vb.mv > @thB + s.ev) OR
   (vb.mv < - @thB + s.ev))
```

In *T2* lines 4-5 expose the ordered index *idx(measuresB.mv)*.

- **Query Q3** identifies two different abnormal behaviors of the same machine at the same time based on two different measurement classes and relative deviations: When and for which machines were the pressure readings of class A higher than *@thA* from its expected value at the same time as the pressure reading of class B were deviating *@thRB %* from its expected value? After the AQIT transformation *Q3* becomes *T3*:

```
Q3:
1 SELECTva.m,greaest(va.bt,vb.bt)
2       least(va.et, vb.et)
3 FROM  measuresA va,measuresB vb,
4       sensor sa, sensor sb
5 WHERE va.m=sa.m AND va.s=sa.s AND
6 vb.m=sb.m AND vb.s=sb.s         AND 7
va.m=vb.m                         AND
8 va.bt<=vb.et AND va.et>=vb.bt AND
9 va.mv - sa.ev > @thA            AND
10 abs((vb.mv-sb.ev)/sb.ev)>@thRB
11
12
13
```

```
T3:
SELECT va.m,greatest(va.bt, vb.bt),
   least(va.et, vb.et)
FROM    measuresA va, measuresB vb,
        sensor sa, sensor sb
WHERE va.m=sa.m AND va.s=sa.s AND
      vb.m=sb.m AND vb.s=sb.s AND
      va.m =vb.m                AND
 va.bt<=vb.et AND va.et>=vb.bt AND
 va.mv >@thA + sa.ev          AND
((vb.mv>(1+@thRB)*sb.ev AND sb.ev >0)
OR (vb.mv<(1+@thRB)*sb.ev AND sb.ev<0)
OR (vb.mv<(-@thRB+1)*sb.ev ANDsb.ev>0)
OR (vb.mv>(-@thRB+1)*sb.ev AND sb.ev<0))
```

Lines 8 in *Q3* selects temporal overlap of the time interval *[va.bt, va.et]* with *[vb.bt, vb.et]*. The functions *greatest(va.bt, vb.bt)* and *least(va.et, vb.et)* return the maximum and minimum values of their two arguments, respectively. These functions are supported by Oracle, MySQL, DB2 and PostgreSQL but not by SQL Server [14]. Therefore, we defined *greatest(x, y)* and *least(x, y)* as user defined functions for SQL Server.

In *T3* line 9 exposes *idx(measuresA.mv)* and lines 10-13 expose *idx(measuresB.mv)*.

4 Stream Log Analysis System (SLAS)

Fig. 1 illustrates the architecture of SLAS. It uses a data stream management system, *DSMS*, to process *raw streams* of measurements from different *machines*. The *log writer* receives from the DSMS a stream of tuples with format *(mc, m, s, ts, mv)* specified as a continuous query. The log writer produces once per system-determined time interval a CSV file of tuples *(m, s, bt, et, mv)* for each measurement class *mc* to be loaded into the corresponding table *measuresMC*. Here, [*bt*,*et*) is the valid time interval for *mv*, computed from *ts*. When the log writer has written a CSV file it notifies the *log loader* for measurement class *mc*, which bulk loads the new log file rows into the corresponding measurement log table *measuresMC*.

In order to limit and customize the amount of log data stored in the DBMS the *log deleter* continuously deletes log data from the DBMS according to user specified configuration parameters.

The user can analyze the stored data streams by issuing historical SQL queries over loaded log data through the *AQIT processor*. The strategy used by AQIT to improve numerical SQL queries is the focus of this paper.

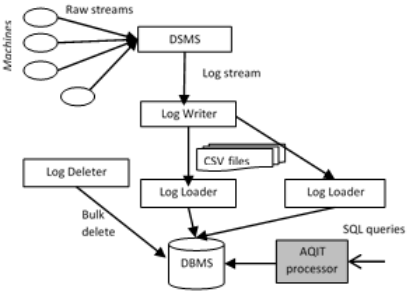

Fig. 1. Stream Log Analyse System

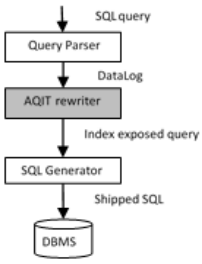

Fig. 2. AQIT Preprocessor

Fig. 2 illustrates the query processing of AQIT. An SQL query is first parsed into an internal query in a Datalog dialect [12]. The *AQIT rewriter* transforms the Datalog query into an equivalent *index exposed query*. The *SQL Generator* transforms the index exposed Datalog query into an equivalent *shipped SQL* query sent to the back-end DBSM through JDBC for optimization and evaluation.

5 Algebraic Query Inequality Transformation

To explain the AQIT transformations we need the following definitions:

Definition 1. A *source predicate r(…)* of a query is a predicate that represents access to a relation named *r*.

Definition 2. If there is a B-tree index *idx(r.a)* on some attribute *a* of a source predicate *r(…a…)*, we say that *r* is an *indexed predicate*.

Definition 3. If there is an occurrence of a variable *v* representing *idx(r.a)* in an indexed predicate *r(…v…)* of a query, we say that *v* is an *indexed variable* in the query.

Definition 4. If there is an inequality $\psi(v,x)$ where *v* is an indexed variable, we say that the indexed variable *v* is *exposed* by the inequality predicate ψ.

In this section, we use *Q1* and *Q2* to show how AQIT works. First the parser translates *Q1*, and *Q2* into the following Datalog queries *DQ1* and *DQ2*:

```
DQ1(m,bt,et) ←
  measuresA(m,s,bt,et,mv)      AND
  sensor(m,s,_,_,ev,_,_)       AND
  v1 = ev + @thA               AND
  mv > v1
```

```
DQ2(m,bt,et) ←
  measuresA(m,s,bt,et,mv)      AND
  sensor(m,s,_,_,ev,_,_)       AND
  v1 = mv - ev                 AND
  v2 = abs(v1)                 AND
  v2 > @thA
```

Here, the source predicates *measuresA(m,s,bt,et,mv)* and *measuresB(m,s,bt.et,mv)* represent relational tables for two different measurement classes. For both tables there

is a B-tree index on *mv* to speed up comparison and proximity queries, and therefore *measuresA()* and *measuresB()* are indexed predicates and the variable *mv* is an indexed variable. In *Q1*, the index *idx(measuresA.mv)* is already exposed because there is a comparison between *measuresA.mv* and variable *v1*, so AQIT will have no effect.

In *Q2*, the index *idx(measuresB.mv)* is not exposed by the inequality predicate *v2 > @thB* since the inequality is defined over a variable *v2*, which is not bound to the indexed attribute *measuresB.mv*. Here AQIT transforms the predicates to expose the index *idx(measuresB.mv)* so in *T2 idx(measuresB.mv)* is exposed in both *OR* branches.

5.1 AQIT Overview

The AQIT algorithm takes a Datalog predicate as input and returns another semantically equivalent predicate that exposes one or several indexes, if possible. AQIT is a *fixpoint* algorithm that iteratively transforms the predicate to expose hidden indexes until no further indexes can be exposed. The full pseudo code can be found in [17].

The transformations are made iteratively by the function *transform_pred()* in Listing 1. At each iteration, it invokes three functions, called *chain()*, *expose()*, and *substitute()*. *chain()* finds some path between an indexed variable and an inequality predicate that can be exposed, *expose()* transforms the found path so that the index becomes exposed, and *substitute()* replaces the terms in the original predicate with the new path.

```
function transform_pred(pred):
input:    A predicate pred
output:   A transformed predicate or the original pred
begin
  if pred is disjunctive then
    set failure = false
    /*result list of transformed branches*/
    set resl = null
    do /*transform each branch*/
      set b = the first not transformed branch in pred
      set nb = transform_pred(b)/*new branch*/
      if  nb not null then add nb to resl
      else  set failure = true
    until failure or no more branch of pred to try
     if not failure then
         /*return a disjunction from resl*/
         return  orify(resl)
     end if
  else if pred is conjunctive then
    set path = chain(pred)
    if path not null then
      set exposedpath = expose(path)
      if  exposedpath not null then
           return substitute(pred, path, exposedpath)
      end if
    end if
  end if
  return pred
end
```

Listing 1. Transform Predicate

Chain. The *chain()* algorithm tries to produce a path of predicates that links one indexed variable with one inequality predicate. If there are multiple indexed variables a simple heuristic is applied. It sorts the indexed variables decreasingly based on selectivities of the indexed attributes, which can be obtained first from the backend DBMS. The path must be a conjunction of *transformable terms* that represent expressions transformable by AQIT. Each transformable term in a path has a single common variable with adjacent terms. Such a chain of connected predicates is called an *index inequality path (IIP)*. Query *DQ2* has the following IIP called *Q2-IIP* from the indexed variable *mv* to the inequality *v2 > @thB*, where the functions '–' and '*abs*' are transformable:

`Q2-IIP: measuresB(m, s, bt, et, mv)→ v1=mv - ev → v2=abs(v1)→ v2>@thB`

In this case *Q2-IIP* is the only possible IIP, since there are no other unexposed index variables in the query after *Q2-IIP* has been formed. The following graph illustrates *Q2-IIP*, where nodes represent predicates and arcs represent the common variable of adjacent nodes:

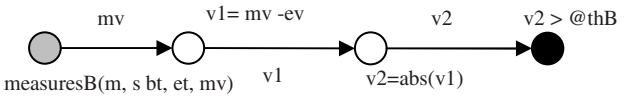

Fig. 3. Q2-IIP

An IIP starts with an indexed *origin* predicate and ends with an inequality *destination* predicate. The origin node in an IIP is always an indexed predicate where the outgoing arc represents one of the indexed variables.

chain() is a backtracking algorithm trying to extend partial IIPs consisting of transformable predicates from an indexed variable until some inequality predicate is reached, in which case the IIP is *complete*. The algorithm will try to find one IIP per indexed variable. If there are several common variables between transformable terms, *chain()* will try each of them until a complete IIP is found. If there are other not yet exposed ordered indexes for some source predicates, the other IIPs may be discovered later in the top level fixpoint iteration.

The *chain()* procedure successively extends the IIP by choosing new transformable predicates *q* not on the partial IIP such that one of *q*'s arguments is the variable of the right-most outgoing arc (*mv* in our case) of the partial IIP. For *DQ2* only the predicate *v1=mv-ev* can be chosen, since *mv* is the outgoing arc variable and '–' is the only transformable predicate in *DQ2* where *mv* is an argument. When there are several transformable predicates, *chain()* will try each of them in turn until the IIP is complete or the transformation fails.

An IIP through a disjunction is treated as a disjunction of IIPs with one partial IIP per disjunct in Listing 1. In this case the index is considered utilized if all partial IIPs are complete.

Expose. The *expose()* procedure is applied on each complete IIP in order to expose the indexed variable. The indexed variable is already exposed if there are no intermediate nodes between the origin node and the destination node in the IIP. For example, the IIP for *Q1* is `Q1-IIP: measuresA(m, s, bt, et, mv)→ mv>v1`. Here the indexed variable *mv* is already exposed to the inequality. Therefore, in this case *expose()* returns the input predicate unchanged.

The idea of *expose()* is to shorten the IIP until the index variable is exposed by iteratively combining the two last nodes through the algebraic rules in

Table 4 into larger destination nodes while keeping the IIP complete. To keep the IIP complete the incoming variable of the last node must participate in some inequality predicate. As an example, the two last nodes in *Q2-IIP* in Fig. 3 are combined into a disjunction in Fig. 4. Here the following algebraic rule is applied: *R10:* $|x| > y \Rightarrow (x > y \vee x < -y)$.

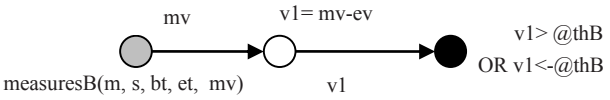

Fig. 4. Q2-IIP after the first reduction

The algebraic rule *R10* exposes a variable *x* hidden inside *abs()* of an inequality. The following table shows how R10 is applied on the two last nodes in Fig. 3 to form the new predicate in Fig. 4.

Table 1. Applying R10

Before	After
v2 = abs(v1) AND v2 > @thB	(v1 > @thB OR v1 < -@thB)

By iteratively exposing each variable on the IIP, the indexed variable (and the index) will possibly be exposed. For example, Q2-IIP in Fig. 4 is reduced into Fig. 5 by applying the algebraic rules *R3:* $x - y > z \Rightarrow x > y + z$ and *R4:* $x - y < z \Rightarrow x < y + z$.

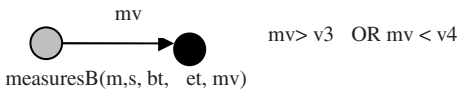

Fig. 5. Q2-IIP after the second reduction

The following two tables show how rules R3 and R4 have been applied:

Table 2. Applying R3

Before	After
v1 = mv –ev AND v1 > @thB	v3 = ev + @thB AND mv > v3

Table 3. Applying R4

Before	After
v1 = mv –ev AND v1 < -@thB	v4 = ev -@thB AND mv < v4

The new variables *v3* and *v4* are created when applying the rewrite rules to hold intermediate values.

In Fig. 5, there are no more intermediate nodes and the index *idx(measuresB.mv)* is exposed, so *expose()* succeeds.

expose() may fail if there is no applicable algebraic rule when trying to combine some two last nodes, in which case the *chain()* procedure will be run again to find a next possible IIP until as many indexed variables as possible are exposed.

Substitute. When *expose()* has succeeded, *substitute()* updates the original predicate by replacing all predicates in the original IIP, except its origin, with the new destination predicate in the transformed IIP [17]. For *Q2* this will produce the final transformed Datalog query:

```
DQ2(m,bt,et)← measuresB(m,s,bt,et,mv)AND
              sensor(m, s, _,_,ev,_,_)   AND
              v3   =  ev + @thB          AND
              v4 = ev -@thB              AND
             (mv < v4 OR mv >  v3)
```

The Datalog query is the translated by the SQL Generator into SQL query *T2*.

5.2 Inequality Transformation Rules

Table 4 the algebraic rewrite rules currently used by AQIT are listed. The list can be extended for new kinds of algebraic index exposures. In the rules, *x, y,* and *z* are variables and ψ denotes any of the inequality comparisons $\geq$, $\leq$,<, or >, while ψ^{-1} denotes the inverse of ψ. *CP* denotes a positive constant (*CP* > *0*), while *CN* denotes a negative constant (*CN* < *0*). Each rule shows how to expose the variable *x* hidden inside an algebraic expression to some inequality expression.

Table 4. Algebraic inequality transformations

R1	$(x + y)\ \psi z$	$\Leftrightarrow$	$x\ \psi (z - y)$
R2	$(y + x)\ \psi z$	$\Leftrightarrow$	$x\ \psi (z - y)$
R3	$(x - y)\ \psi z$	$\Leftrightarrow$	$x\ \psi (z + y)$
R4	$(y - x)\ \psi z$	$\Leftrightarrow$	$x\ \psi^{-1} (y - z)$
R5	$(x * CP)\ \psi z$	$\Leftrightarrow$	$(x\ \psi z/CP)$
R6	$(x * CN)\ \psi z$	$\Leftrightarrow$	$(x\ \psi^{-1}\ z/CN)$
R7	$x/y\ \psi z \wedge\ y != 0$	$\Leftrightarrow$	$(x\ \psi y*z \wedge y > 0) \vee (x\ \psi^{-1} z*y \wedge y < 0)$
R8	$y/x\ \psi z$	$\Leftrightarrow$	$(y/z\ \psi x \wedge x*z > 0) \vee (y/z\ \psi^{1} x \wedge x*z < 0)$ $\vee\ (y = 0 \wedge 0\ \psi\ z)$
R9	$/x/ \leq y$	$\Leftrightarrow$	$(x \leq y \wedge x\ \geq - y)$
R10	$/x/ \geq y$	$\Leftrightarrow$	$(x\ \geq y \vee x \leq - y)$
R11	$\sqrt{x}\ \psi y$	$\Leftrightarrow$	$x\ \psi y^2$
R12	$x^y\ \psi z$	$\Leftrightarrow$	$(x\ \psi\ \sqrt[y]{z} \wedge y\ > 0) \vee\ (x\ \psi^{-1}\ \sqrt[y]{z}\ \wedge y < 0)$ $\vee\ (x\ \psi z \wedge y = 0)$
R13	$(x+ y)/x\ \psi\ z$	$\Leftrightarrow$	$(1+ y/x)\ \psi\ z$
R14	$/(x - y) / y / > z$	$\Leftrightarrow$	$(x > (z + 1)* y \wedge y > 0) \vee (x < (z + 1)* y \wedge\ y < 0)$ $\vee (x < (- z+ 1)* y \wedge y > 0) \vee (x > (- z + 1)* y \wedge\ y < 0)$

6 Experimental Evaluation

We experimentally compared the performance of a number of typical queries finding different kinds of abnormalities based on 16000 real log files from two industrial machines. To simulate data streams from a large number of machines, 8000 log files were constructed by pairing the real log files two-by-two and then time-stamping their events based on off-sets from their first time-stamps. This produces realistic data logs and enables scaling the data volume by using an increasing number of log files.

6.1 Setup

To investigate the impact of AQIT on the query execution time, we run the SLAS system with SQL Server™ 2008 R2 as DBMS on a separate server node. The DBMS was running under Windows Server 2008 R2 Enterprise on 8 processors of AMD Opteron ™ Processor 6128, 2.00 GHz CPU and 16GB RAM. The experiments were conducted with and without AQIT preprocessing.

6.2 Data

Fig. 6 (a) is a scatter plot from a small sampled time interval of pressure readings of class A. This is an example of an asymmetric measurement series with an initial warm-up period of 581.1 seconds.

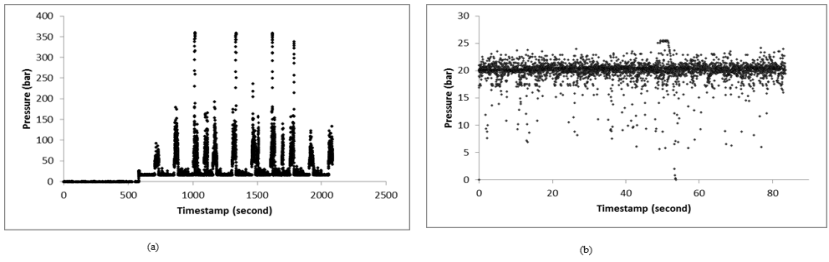

Fig. 6. Pressure measured of class A (a) and class B (b)

The abnormal behavior in this case is that the measured values are larger than the expected value (17.02) within a threshold. When the deviation threshold is 0 all measurements are abnormal, while when the threshold is 359.44 no measurements are abnormal. For example, *Q1* finds when a sensor reading of class A is abnormal based on threshold *@thA* that can be varied.

Fig. 6 (b) plots pressure readings of measurements of class B over a small sampled time interval. Here the abnormality is determined by threshold *@thB,* indicating absolute differences between a reading and the expected value (20.0), as specified in *Q2*. When the threshold is 0 all measurements are abnormal, while when the threshold is 20.0 no measurements are abnormal.

In addition, the abnormality of measurements of class B is determined by threshold *@thRB* as in *Q3*, indicating relative difference between a reading and the expected

value. When the relative deviation threshold is 0%, no measurements are abnormal, while when the threshold is 100% all measurements are abnormal.

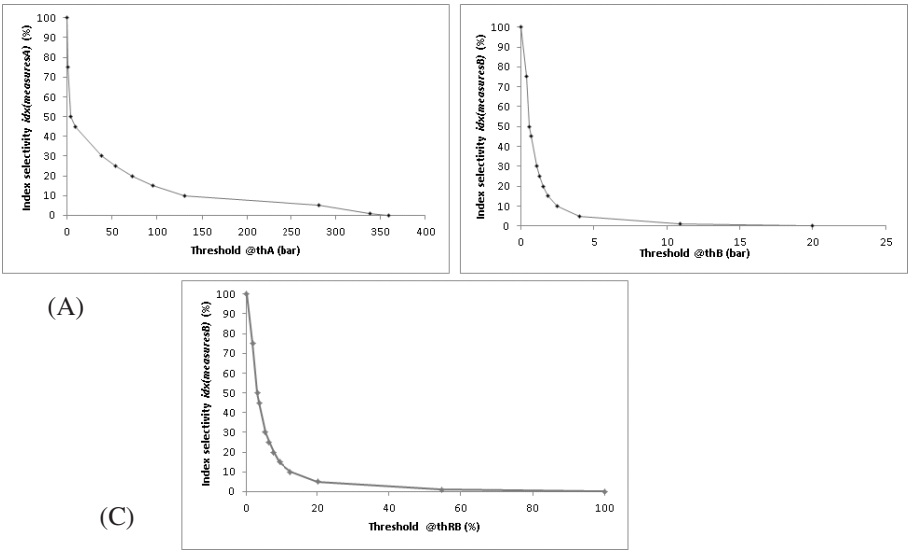

Fig. 7. Thresholds and selectivity mappings

6.3 Benchmark Queries

We measured the impact of index utilization exposed by AQIT by varying the abnormality thresholds *@thA* for queries determining deviations in *measuresA,* and the thresholds *@thB* and *@thRB* for queries determining deviations in *measuresB*. The larger the threshold values the fewer abnormalities will be detected. We also defined three other benchmark queries *Q4*, *Q5*, and *Q6*. All the detailed SQL and Datalog formulations before and after AQIT for the benchmark queries are listed in [18].

- *Q4* identifies when the pressure readings of class B deviates more than *@thB* for the machines in a list *machine-models* of varying length. Here, if a query spans many machine models the impact of AQIT should decrease since many different index keys are accessed.

```
Q4: SELECT vb.m, vb.bt, vb.et
FROM measuresB vb, sensor s,
     machine ma
WHERE vb.m = s.m AND va.s=s.s AND
    vb.m = ma.m                AND
    ma.mm in@machine-models    AND
    abs(vb.mv - s.ev) > @thB
```

```
T4: SELECT vb.m, vb.bt, vb.et
FROM measuresB vb, sensor s,
     machine ma
WHERE vb.m = s.m AND va.s=s.s AND
      vb.m = ma.m              AND
      ma.mm in @machine-models AND
(vb.mv > @thB + s.ev OR vb.mv < -
@thB + s.ev
```

- *Q5* identifies when the pressure reading of class B deviates more than *@thB* for two specific machine models using a temporal join. The query involves numerical expressions over two indexed variables, which are both exposed by AQIT. See [18] for details.
- Query *Q6* is a complex query that identifies a sequence of two different abnormal behaviors of the same machine happening within a given time interval, based on two different measurement classes: On what machines the pressure readings class B were out-of-bounds more than *@thB* within 5 seconds after the pressure readings of class A were higher than *@thA* from the expected value. Here, both *idx(measuresA.mv)* and *idx(measuresB.mv)* are exposed by AQIT. See [18] for details.

6.4 Performance Measurements

To measure performance based on different selectivities of indexed attributes, in Fig. 7 we map the threshold values to the corresponding measured index selectivities of *idx(measuresA.mv)* and *idx(measuresB.mv)*. 100% of the abnormalities are detected when any of the thresholds is 0 and thresholds above the maximum threshold values (*@thA*=359.44, *@thB*=20.0, and *@thRB*=100%) detect 0% abnormalities.

Experiment A varies the database size from 5GB to 25GB while keeping the selectivities (abnormality percentages) at 5% and a list of three different machine models in *Q4*.

Fig. 8 (a) shows the performance of example queries *Q2, Q3, Q4, Q5*, and *Q6* (without AQIT) and their corresponding transformed queries *T2, T3, T4, T5*, and *T6* (with AQIT) when varying the database size from 5 to 25 GB. The original queries without AQIT are substantially slower since no indexes are exposed and the DBMS will do full scans, while for transformed queries the DBMS backend can utilize the exposed indexes.

Experiment B varies index selectivities of *idx(measuresA.mv)* and *idx(measuresB.mv)* while keeping the database size at 25 GB and selecting three different machine models in *Q4*. We varied the index selectivities from 0% to 100%. Fig. 8 (b) presents execution times of the all benchmark queries with and without AQIT.

Without AQIT, the execution times for *Q2* - *Q6* stay constant when varying the selectivity since no index is utilized and the database tables are fully scanned.

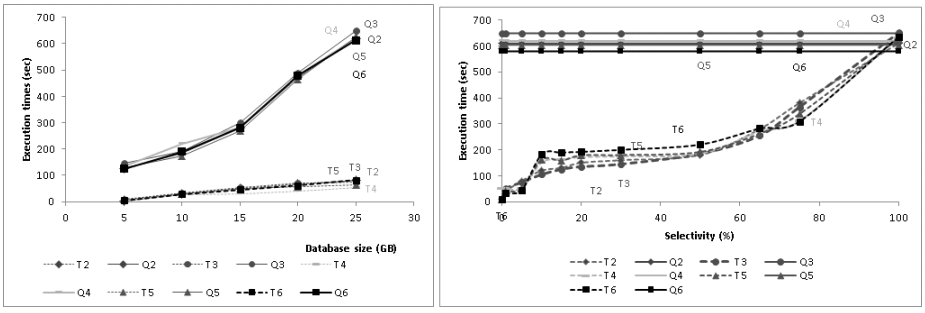

Fig. 8. All queries while changing DB size (a) and selectivities (b)

Fig. 8 (b) shows that AQIT has more effect the lower the selectivity, since index scans are more effective for selective queries. For non-selective queries the indexes are not useful. When all rows are selected the AQIT transformed queries are slightly slower than original ones; the reason being that they are more complex. In general AQIT does not make the queries significantly slower.

Experiment C varies the number machine models in *Q4* from 0 to 25 while keeping the database size at 25 GB and the selectivity at 5%, as illustrated by Fig. 9. It shows that when the list is small the transformed query *T4* scales much better than the original query *Q4*. However, when the list of machine increases, *T4* is getting slower. The reason is that the index *idx(measuresB.mv)* is accessed once per machine model, which is faster for fewer models.

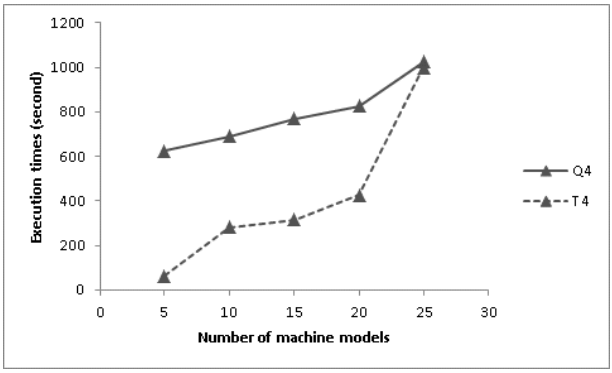

Fig. 9. Execution times of Query 4 when varying the list of machine models

The experiments A, B, and C show that AQIT improves the performance of the benchmark queries substantially and will never make the queries significantly slower. In general AQIT exposes hidden indexes while the backend DBMS decides whether to utilize them or not.

7 Conclusion and Future Work

In order to improve the performance of queries involving complex inequality expression, we investigated and introduced the general algebraic query transformation algorithm AQIT. It transforms a class of SQL queries so that indexes hidden inside numerical expressions are exposed to the back-end query optimizer.

From experiments, which were made on a benchmark consisting of real log data streams from industrial machines, we showed that the AQIT query transformation substantially improves query execution performance.

We presented our general system architecture for analyzing logged data streams, based on bulk loading data streams into a relational database. Importantly, looking for abnormal behavior of logged data streams often requires inequality search conditions and AQIT was shown to improve the performance of such queries.

We conclude that AQIT improves substantially the query performance by exposing indexes without making the queries significantly slower.

Since inequality conditions also appear in spatial queries we plan to extend AQIT to support transforming spatial query conditions as well user defined indexing. We also acknowledge that the inequality conditions could be more complex with multiple variables and complex mathematical expression, which will require other algebraic rules.

Acknowledgements. The work was supported by the Smart Vortex EU project [15].

References

1. Andrew, A.D., Cain, G.L., Crum, S., Morley, T.D.: Calculus Projects Using Mathematica. McGraw-Hill (1996)
2. Arvind, T., Samuel, M.: Querying continuous functions in a database system. In: Proc. SIGMOD 2008, Vancouver, Canada, pp. 791–804 (2008)
3. Celko, J.: SQL for Smarties, 4th edn. Advanced SQL Programming (2011) ISBN: 978-0-12-382022-8
4. Chang, C.M.: Mathematical Analysis in Engineering. Cambridge University Press (1994)
5. Dageville, B., Das, D., Dias, K., Yagoub, K., Zaït, M., Ziauddin, M.: Automatic SQL Tuning in Oracle 10g. In: Proc. VLDB 2004, Toronto, Canada, pp. 1098–1109 (2004)
6. Eltabakh, M.Y., Eltarras, R., Aref, W.G.: Space-Partitioning Trees in PostgreSQL: Realization and Performance. In: Proc ICDE, Atlanta, Georgia, USA, pp. 100–112 (April 2006)
7. Gabriel, K., Leonid, L., Jan, P.: Constraint Databases, pp. 21–54. Springer, Heidelberg, ISBN 978-3-642-08542-0
8. Gray, J., Szalay, A., Fekete, G.: Using Table Valued Functions in SQL Server 2005 to Implement a Spatial Data Library,Technical Report, Microsoft Research Advanced Technology Division (2005)
9. Grumbach, S., Rigaux, P., Segoufin, L.: The DEDALE system for complex spatial queries. In: Proc SIGMOD 1998, Seattle, Washington, pp. 213–224 (1998)
10. Hwang, D.J.-H.: Function-Based Indexing for Object-Oriented Databases, PhD Thesis, Massachusetts Institute of Technology, 26–32 (1994)
11. Leccotech. LECCOTECH Performance Optimization Solution for Oracle, White Paper (2003), `http://www.leccotech.com/`
12. Litwin, W., Risch, T.: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. IEEE Transactions on Knowledge and Data Engineering 4(6) (December 1992)
13. Oracle Inc. Query Optimization in Oracle Database 10g Release 2. An Oracle White Paper (June 2005)
14. Snodgrass, R.T.: Developing Time-Oriented Database Applications in SQL. Morgan Kaufmann Publishers, Inc., San Francisco (1999) ISBN 1-55860-436-7
15. Smart Vortex Project , `http://www.smartvortex.eu/`
16. Quest Software.Quest Central for Oracle: SQLab Vision (2003), `http://www.quest.com`
17. `http://www.it.uu.se/research/group/udbl/aqit/PseudoCode.pdf`
18. `http://www.it.uu.se/research/group/udbl/aqit/Benchmark_queries.pdf`