# Rephrasing Rules for Off-The-Shelf SQL Database Servers

Ilir Gashi, Peter Popov
*Centre for Software Reliability, City University,*
*Northampton Square, London, EC1V 0HB*
I.Gashi@city.ac.uk, ptp@csr.city.ac.uk

## Abstract

*We have reported previously [1] results of a study with a sample of bug reports from four off-the-shelf SQL servers. We checked whether these bugs caused failures in more than one server. We found that very few bugs caused failures in two servers and none caused failures in more than two. This would suggest a fault-tolerant server built with diverse off-the-shelf servers would be a prudent choice for improving failure detection. To study other aspects of fault tolerance, namely failure diagnosis and state recovery, we have studied the "data diversity" mechanism and we defined a number of SQL rephrasing rules. These rules transform a client sent statement to an additional logically equivalent statement, leading to more results being returned to an adjudicator. These rules therefore help to increase the probability of a correct response being returned to a client and maintain a correct state in the database.*

## 1. Introduction

Fault tolerance is frequently the only viable approach of obtaining the required system dependability from systems built out of "off-the-shelf" (OTS) products [2]. There are various methods in which this fault tolerance can be achieved ranging from simple error detection and recovery add-ons (e.g. wrappers [3]) to diverse redundancy replication using diverse versions of the components.

These design solutions are well known. Questions remain, however, about the dependability gains and implementation difficulties for a specific system.

We have studied some of these issues in SQL database servers, a very complex category of off-the-shelf products. We have previously reported [1] results from a study with a sample of bug reports from four off-the-shelf SQL servers so as to assess the possible advantages of software fault tolerance - in the form of modular redundancy with diversity - in complex off-the-shelf software. We found that very few bugs cause failures in two servers and none cause failures in more than two, which would indicate that significant dependability improvements can be expected from the deployment of a fault-tolerant server built out of diverse off-the-shelf servers in comparison with individual servers or the non-diverse replicated configurations.

Although we found that using multiple diverse SQL servers can dramatically improve error detection rates it does not make them 100%, e.g. our study [1] found four bugs causing identical non-self-evident failures in two servers. Thus there is room for improving failure detection further. Many of the cases, in which a failure was detected did not allow for immediate diagnosis of the failed server. Fault tolerance requires also diagnosing the faulty server and maintaining data consistency among the databases in addition to failure detection. To improve the situation, we studied the mechanism called "data diversity" by Ammann and Knight [4] (who studied it in a different context). The simplest example of the idea in [4] refers to computation of a continuous function of a continuous parameter. The values of the function computed for two close values of the parameter are also close to each other. Thus, failures in the form of dramatic jumps of the function on close values of the parameter can not only be detected but also corrected by computing a "pseudo correct" value. This is done by trying slightly different values of the parameter until a value of the function is calculated which is close to the one before the failure. This was found [4] to be an effective way of detecting as well as masking failures, i.e. delivering fault-tolerance. Data diversity, thus, can help with failure detection and state recovery, and thus complement fault-tolerance solutions which employ diverse modular redundancy, as well as helping achieve a certain degree of fault tolerance without employing diverse modular redundancy.

Data diversity is applicable to SQL servers because of the inherent redundancy that exists in the SQL language: statements can be "rephrased" into different, but logically equivalent [sequences of] statements. While working with the bug reports we found examples where a particular statement causes a failure in a server but a rephrased version of the same statement does not. Examples of such statements often appear in bug reports as "workarounds".

In this paper we provide details of how SQL rephrasing can be employed systematically in a fault-tolerant server and provide examples of useful rephrasing rules. We also report on performance measurements using the TPC-C [5] benchmark client implementation to get some initial estimates of the delays introduced by rephrasing.

The paper is structured as follows: in section 2 we give details of the architecture of a fault-tolerant server employing rephrasing. In section 3 we give details of the data diversity study we have conducted for defining SQL rephrasing rules and illustrate how one of these rules has been used as a workaround for two known bugs of two SQL servers. In section 4 we give some empirical results of experiments we have conducted to measure the performance penalty due to rephrasing. In section 5 we discuss some general implications of our results and finally in section 6 some conclusions are presented with possibilities for further work.

## 2. Architecture of a Fault-Tolerant Server

### 2.1 General Scheme

Data replication is a well-understood subject [6], [18], [7]. The main problem replication protocols deal with is guaranteeing consistency between copies of a database without imposing a strict synchronisation regime between them. A study which compared various replication protocols in terms of their performance and the feasibility of their implementation can be found in [8]. Existing protocols implement efficient solutions for this problem, but depend on running copies of the *same* (non-diverse) server. These schemes would not tolerate non-self-evident[1] failures that cause incorrect writes to the database or that return incorrect results from read statements. For the former, incorrect writes would be propagated to the other replicas and for the latter, incorrect results would be returned to the client. This deficiency can be overcome by building a fault-tolerant server node ("FT-node") from two or more diverse SQL servers, wrapped together with a "middleware" layer to appear to each client as a single SQL server. An illustration of this architecture with two diverse Off-The-Shelf servers ("O-servers") is shown in Fig. 1. A brief explanation of the figure follows. Several nodes (computers) are depicted which run client applications (Client node 1, Client node 2 and Client node 3) or server applications (Middleware node, RDBMS 1 node and RDBMS 2 node). The bottom three nodes together form the FT-server. Components may share a node: e.g. Replication Middleware, and the two SQL connectors for dialects 1 and 2 are deployed on the Middleware node. The SQL connectors additionally contain the SQL rephrasing rules. The diagram assumes that the Off-The-Shelf servers (O-servers) run on separate nodes, RDBMS 1 node and RDBMS 2 node. The circles represent the interfaces through which the components interact. Each SQL connector, implements the SQL Connector API interface used by the Replication Middleware component. This, in turn implements the Middleware API interface via which the client applications access the FT-server, either directly or via a driver for the FT-server in a specific run-time environment, e.g. JDBC driver or .NET Provider.

Further improvements to this architecture would be to also run diverse replicas of the middleware component. We have described elsewhere [9], [2] in more detail the FT-node architecture. Here we will only elaborate on the parts relevant to the discussion of rephrasing.

### 2.2 SQL Connectors

The O-servers are not fully compatible: they "speak different dialects" of SQL, despite being compliant at various levels with SQL standards. Therefore the FT-server includes a translator between these dialects, defined for a subset of SQL (e.g. "SQL-92 entry level") plus some more advanced features important for enterprise applications (such as TRIGGERs and STORED PROCEDUREs). The translators are depicted as "SQL Dialect Connector's" in Fig 1.

A similar idea (implemented in [10], [11]) is to redefine the grammar of one database server to make it compatible with that of another while keeping the core database engine unchanged.

[1] In [1] we classified the failures according to their detectability by a client of the database servers into: *Self-Evident failures* - engine crash failures, cases in which the server signals an internal failure as an exception (error message) and performance failures; *Non-Self-Evident failures*: incorrect result failures, without server exceptions within an accepted time delay.

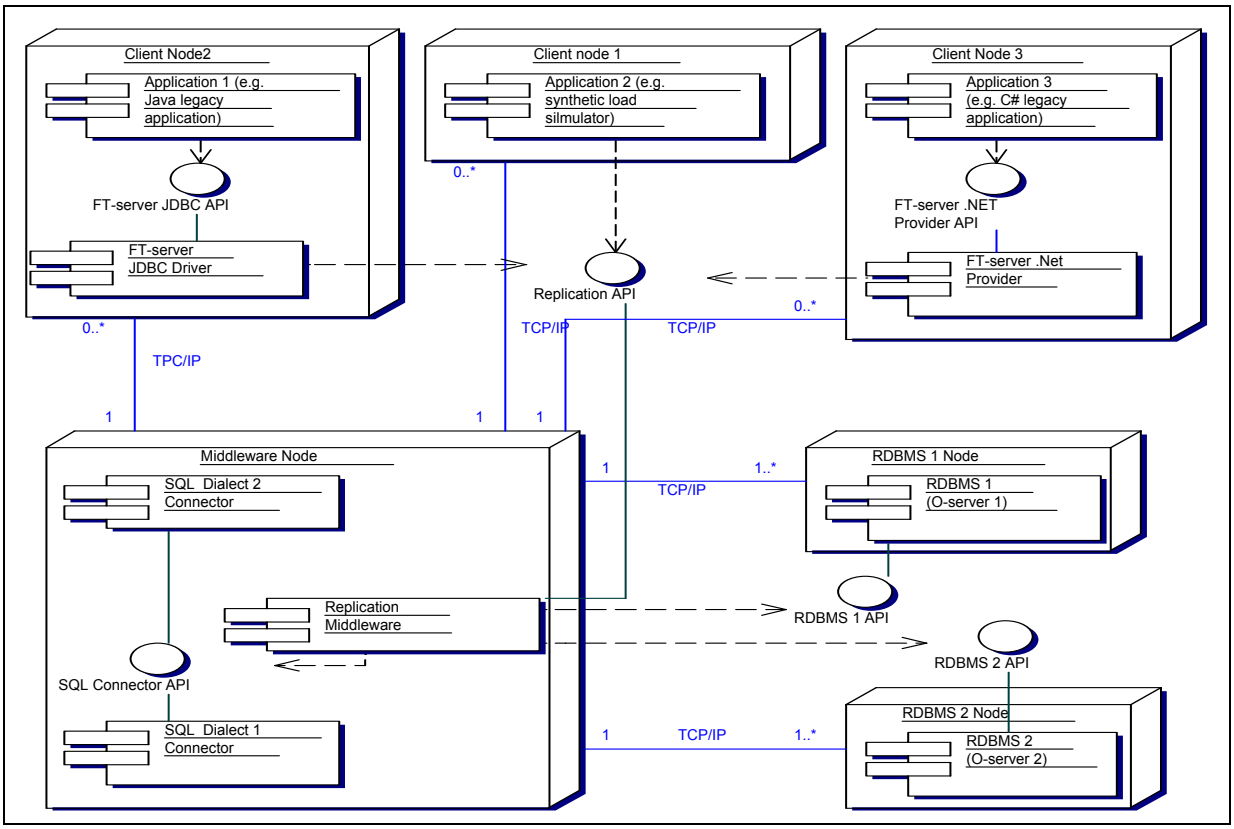


**Fig. 1 - UML Deployment diagram of the FT-server.**

## 2.3 Failure Detection, Masking, Recovery

The middleware of the FT-server includes extensive functionality for failure detection, masking and state recovery. Self-evident server failures are detected as in a non-diverse server, via server error messages (i.e. via the existing error detection mechanisms inside the servers), and time-outs for crash and performance failures. Diversity gives the additional capability of detecting non-self-evident failures by comparing the outputs[2] of the different O-servers. In a FT-node with 3 or more diverse O-servers, majority voting can be used to choose a result and thus mask the failure to the clients, and identify the failed O-server which may need a recovery action to correct its state. With a 2-diverse FT-node, if the two O-servers give different results, the middleware cannot decide which O-server is in error. This is where "data diversity" can help by providing additional results to break the tie (more in the next subsection). State recovery of the database can be obtained in the following ways:

- via standard backward error recovery, which will be effective if the failures are due to transient failures (caused by so called "Heisenbugs" [12]). To command backward error recovery, the middleware may use the standard database transaction mechanisms: aborting the failed transaction and replaying its statements may produce a correct execution. With "data diversity" a finer granularity level of recovery is possible using SAVEPOINTs and ROLLBACKs;
- additionally, diversity offers ways of recovering from non-transient failures (caused by so called "Bohrbugs" [12]), by essentially copying the database state of a correct server into the failed one (similarly to [13]). Since the formats of the database files differ between the servers, the middleware would need to query the correct server[s] for their database contents and command the failed server to write them into the corresponding records in its database, similar to what is proposed in [14]. This

[2] An "output" may be the results from a SELECT statement or the number of rows affected for a write (INSERT, UPDATE and DELETE) statement. For INSERT and UPADTE statements a more refined way would be to read back the affected rows and use those for comparison.

would be expensive, perhaps to be completed off-line, but a designer can use multi-level recovery, in which the first step is to correct only those records that have been found erroneous on read statements.

### 2.4 Data Diversity Extensions

Even with just two diverse O-servers, many of the O-server failures may be masked by using "data diversity" (rephrasing an SQL statement into a different, but semantically equivalent one) to solicit "second opinions" from the O-servers and if possible outvote the incorrect response.

Data diversity could be implemented via an algorithm in the "Middleware Node" that rephrases statements according to predefined rules. We can define these rules for each type of SQL statement defined by the SQL grammar implemented by the server. These rules therefore may form part of the "SQL Dialect Connectors". Upon receiving a statement from a client application the middleware can look up a rule from the list of available rules and rephrase the statement. The middleware must allow for new rules to be defined as and when necessary. If the middleware exhausts the list of rules that it can apply to a certain statement but no "correct result"[3] can be established by applying the closed adjudication mechanism then an error message is returned to the client.

Data diversity can be used *with* or *without* design diversity. Architectural schemes using data diversity are similar to those using design diversity. For instance, Amman and Knight in [4] describe two schemes, which they call "retry block" and "n-copy programming", which can also be used for SQL servers. The "retry block" is based on backward recovery. A statement is only rephrased if either the server "fail-stops" or its output fails an acceptance test. In "n-copy programming", a copy of the statement as issued by the client is sent to one of the O-servers and rephrased variant(s) are sent to the others; their results are voted to mask failures.

Data diversity allows for a finer-granularity of state recovery, which is facilitated by the implementation of "SAVEPOINT" and "ROLLBACK" within transactions. The procedure (written in pseudocode), for a statement within a transaction, is given at the end of this subsection.

A performance optimization could be to perform adjudication at an intermediate step of the WHILE loop execution rather than at the end (e.g. for a "majority voting" adjudication, if there are five rules for a particular statement then could check after the execution of the first three rephrased versions of the statement whether results returned by each of them are identical; if yes then majority result is already obtained and therefore no need for the last two rephrased versions of the statement to be executed).

The SAVEPOINT and ROLLBACK approach is the correct way of ensuring the "isolation" property of an ACID transaction.[4] Otherwise, if we "ABORTed" the transaction and started a new one to perform the rephrased version of the statement, a concurrent transaction may have updated rows in the target table. This would lead to different results being returned by the O-server for the rephrased statement even though the behavior is not faulty.

```
WHILE more rephrasing rules available for the statement DO
    IF WRITE (i.e. DML (INSERT, UPDATE or DELETE) or DDL (e.g.
    CREATE VIEW etc.)) statement THEN
        SAVEPOINT;
        Execute WRITE statement[s] produced by the current re-
        phrasing rule;
        READ the rows amended by the WRITE statement;
        Store the results produced by the preceding READ state-
        ment;
        ROLLBACK TO last SAVEPOINT;
    ELSE IF READ (i.e. SELECT) statement THEN
        Execute READ statement[s] produced by the current re-
        phrasing rule;
        Store the results produced by the READ statement;
    END IF
END WHILE
Adjudicate from the stored results produced by each rephrased version
of the statement;
IF adjudication succeeds (e.g. "majority voting" produced a result) THEN
    Execute the statement which was adjudicated to be correct;
ELSE
    ABORT current Transaction
    Raise an exception;
END IF
```

## 3. SQL Rephrasing Rules

As explained in section 2, the support for data diversity can be implemented in the middleware in the form of rephrasing rules. The initial step is defining the rules that are to be implemented. The rules can be defined by studying in depth the SQL language itself to identify the parts of the language which are synonymous and therefore enable the definition of logically equivalent rephrasing rules. We took a different more

[3] Depending on the setup used a correct result could be either the majority result or one that passes an acceptance test.

[4] This is under the assumption that the ACID property of the transaction is failure-free.

direct approach to defining these rules: we studied the known bugs reported for 4 open-source servers, namely Interbase 6.0, Firebird 1.0[5], PostgreSQL 7.0 and PostgreSQL 7.2 (abbreviated IB 6.0, PG 7.0, FB 1.0 and PG 7.2 respectively). However our intention was not to simply define workaround rules which are highly bug specific, but instead to define generic rephrasing rules, which can be used in a broader setting. As a result we found that some of the generic rules that we defined could be applied to multiple bugs in our study. We provide examples next.

### 3.1 Generic Rules

The "generic rules" are rephrasing rules, which can be applied to a range of 'similar' statements, be it DML (data manipulation language: SELECT, INSERT, UPDATE and DELETE) or DDL (data definition language e.g. CREATE TABLE etc.) statements. We have defined a total of 14 generic rephrasing rules. Full details of these rules are in [15]. We will provide details of Rule 8 and how it proved to be a useful *workaround* for two different bugs reported for two different servers.

Rule 8: *An SQL* VIEW *can be rephrased as an SQL* STORED PROCEDURE *or SQL* TEMPORARY TABLE

This rule proved to be a useful workaround for FB 1.0 Bug 488343 [16]. To observe the failure the bug report details the following setup:

```
CREATE TABLE CUSTOMERS (ID INT, NAME, VARCHAR(10) );
CREATE TABLE INVOICES (ID INT, CUST_ID INT, CODE
  VARCHAR(10), QUANTITY INT);
INSERT INTO CUSTOMERS VALUES (1, 'ME');
INSERT INTO INVOICES VALUES (1, 1, 'INV.1', 5);
INSERT INTO INVOICES VALUES (2, 1, 'INV.2', 10);
INSERT INTO INVOICES VALUES (3, 1, 'INV.3', 15);
INSERT INTO INVOICES VALUES (4, 1, 'INV.4', 20);
```

The following VIEW is faulty (specifically, the use of the SQL DISTINCT keyword to filter the results of a SELECT statement is faulty in SQL VIEWs of the FB 1.0 server):

```
CREATE VIEW V_CUSTOMERS AS SELECT DISTINCT ID, NAME
  FROM CUSTOMERS;
```

The failure can be observed by issuing the following statement:

```
SELECT SUM(INV.QUANTITY) FROM INVOICES INV INNER JOIN
  V_CUSTOMERS CUST ON INV.CUST_ID = CUST.ID;
SUM
20
```

[5] Firebird is the open-source descendant of Interbase 6.0. The later releases of Interbase are issued as closed-development by Borland.

The expected result is 50 not 20. If we use a STORED PROCEDURE instead of the VIEW then the correct results is returned[6]:

```
SET TERM !!;
CREATE PROCEDURE V_CUSTOMERS RETURNS  (ID INT, NAME
VARCHAR(10)) AS
BEGIN
        FOR SELECT DISTINCT ID, NAME FROM CUSTOMERS
        INTO :ID, :NAME DO
                BEGIN
                        SUSPEND;
                END
END!!
SET TERM; !!
```

Issuing the same SELECT statement as before we obtain the expected result (50):

```
SELECT SUM(INV.QUANTITY) FROM INVOICES INV INNER JOIN
  V_CUSTOMERS CUST ON INV.CUST_ID = CUST.ID;
SUM
50
```

The same rule was a useful workaround for another bug, this time the PG 7.0 bug 23 [17]. To observe the failure the bug report details the following setup:

```
CREATE TABLE L (PID INT NOT NULL, SEARCH BOOL, SERVICE
  BOOL);
INSERT INTO L VALUES (1,'T','F'); INSERT INTO L VALUES (1,'T','F');
INSERT INTO L VALUES (1,'T','F'); INSERT INTO L VALUES (1,'T','F');
INSERT INTO L VALUES (1,'T','F'); INSERT INTO L VALUES (1,'F','F');
INSERT INTO L VALUES (1,'F','F'); INSERT INTO L VALUES (2,'F','F');
INSERT INTO L VALUES (3,'F','F'); INSERT INTO L VALUES (3,'T','F');
```

The following VIEWs are then defined (notice the use of the GROUP BY clause):

```
CREATE VIEW CURRENT AS SELECT PID, COUNT(PID), SEARCH,
  SERVICE FROM L GROUP BY PID, SEARCH, SERVICE;
CREATE VIEW CURRENT2 AS SELECT PID, COUNT (PID),
  SEARCH, SERVICE FROM L GROUP BY PID, SEARCH, SERVICE;
```

By issuing the following SELECT statement incorrect results are obtained (this is due to the GROUP BY clause used in the VIEWs and the COUNT used on a column from a VIEW):

```
SELECT CURRENT.PID, CURRENT.COUNT AS SEARCHTRUE,
  CURRENT2.COUNT AS
SEARCHFALSE FROM CURRENT,CURRENT2 WHERE
  CURRENT.PID =CURRENT2.PID AND CURRENT.SEARCH='T'
  AND CURRENT2.SEARCH='F' AND CURRENT.SERVICE='F' AND
  CURRENT2.SERVICE='F';
-- pid | searchtrue | searchfalse
--   1 |       10   |       10
--   3 |       1    |       1
```

The expected results are:

```
-- pid | searchtrue | searchfalse
--   1 |       5    |       2
--   3 |       1    |       1
```

[6] The syntax used is specific for Firebird.

By using TEMPORARY TABLEs instead of VIEWs the correct result is obtained:

```
SELECT PID, COUNT(PID), SEARCH, SERVICE INTO TEMP
    CURRENT FROM L GROUP BY PID, SEARCH, SERVICE;
SELECT PID, COUNT(PID), SEARCH, SERVICE INTO TEMP
    CURRENT2 FROM L GROUP BY PID, SEARCH, SERVICE;
SELECT CURRENT.PID,CURRENT.COUNT AS SEARCHTRUE,
    CURRENT2.COUNT AS SEARCHFALSE FROMCURRENT,
    CURRENT2 WHERE CURRENT.PID=CURRENT2.PID AND
    CURRENT.SEARCH='T' AND CURRENT2.SEARCH='F' AND
    CURRENT.SERVICE='F' AND CURRENT2.SERVICE='F';
-- pid | searchtrue | searchfalse
--  1 |      5     |      2
--  3 |      1     |      1
```

We used TEMPORARY TABLEs in PG 7.0 and not STORED PROCEDUREs since PG 7.0 does not support functions (procedures) that return multiple rows.

Details of the other generic rephrasing rules and how they can be used as workarounds for other reported bugs are given here [15].

We looked at how many of the generic rules can be applied to the bugs reported for the open-source servers in our bugs study. The results are shown in Table 1. The leftmost three columns of the table show the results for the non-self-evident failures caused by read (i.e. SELECT) statements. Clearly, a number of these are also classified as a "user error", i.e. the user issues an incorrect statement, which the server incorrectly executes without raising an exception. For example IB 6.0 incorrectly executes a statement such as SELECT X FROM A, B even though the column X is defined in both tables A and B, which can lead to ambiguous results. PG 7.0 / PG 7.2, correctly, raise an exception.

If we take away the "user error" bugs then we can see that in all the server pairs the generic rules can be used as workarounds for at least 80% of the non-self-evident failures caused by read statements.

The right-most 4 columns of the table are for the bugs that cause state-changing failures, which have been further subdivided into bugs in DDL and write statements. We can see that generic rules can be used as workarounds for at least 60% of failures caused by the state-changing statements.

### 3.2 Specific Rules

The generic rephrasing rules that we have defined do not provide workarounds for all the failures caused by the bugs collected in our study. For these failures specific workaround rules need to be defined. For example recursive BEFORE UPDATE TRIGGERS can return error messages in FB 1.0/IB 6.0 which means the table for which the trigger is defined becomes unusable (FB 1.0 bug 625899 [16]). A generic rule could not be defined for this bug. A specific workaround (and a generic recovery procedure) upon encountering this error message would be to:

- disable the trigger in FB 1.0 / IB 6.0
- read the log of the other server to check the sequence of the write statements that have been issued as a result of the trigger
- send this sequence of statements explicitly to the FB 1.0 / IB 6.0 server

The workaround above would work in a diverse server-type configuration if the other server[s] works correctly (the other server[s] in our study do not contain this bug) while without design diversity a fault, clearly, cannot be dealt with this way.

We have found that a large number of bugs, if server diversity is not employed, would require very specific rules to be defined to workaround the failures that they cause. In many cases these rules require substantial new implementation in the form of "wrapping" of the results returned to the client (or for write statements before they are stored in the database) or re-implementing parts of the functionality of the database that are found to be faulty and no workaround exists in SQL. Although possible such an approach is clearly limited because the newly developed code can itself be faulty which may diminish the gains in reliability that can be obtained from its use. This reiterates that design diversity is desirable.

## 4. Performance Implications of Rephrasing

To measure the performance implications of rephrasing, we conducted a number of experiments based

**Table 1. A summary of applying the generic rephrasing rules for non-self evident and state-changing bugs of IB 6.0 and PG 7.0 and the later releases FB 1.0 and PG 7.2**

| Server pair | Non-self evident non-state-changing failures (SELECT statements) | | | State-changing failures | | | |
|---|---|---|---|---|---|---|---|
| | | | | DDL statement failures | | Write statement failures | |
| | Total | Total covered by generic rules | Total user errors * | Total | Total covered by generic rules | Total | Total covered by generic rules |
| IB 6.0 + PG 7.0 | 21 | 12 | 6 | 21 | 13 | 9 | 7 |
| IB 6.0 + PG 7.2 | 26 | 18 | 6 | 19 | 13 | 7 | 5 |
| FB 1.0 + PG 7.0 | 16 | 11 | 2 | 19 | 13 | 8 | 6 |
| FB 1.0 + PG 7.2 | 19 | 15 | 2 | 17 | 13 | 6 | 4 |

on the industry standard benchmark for databases - TPC-C [5][7]. The factors which degrade performance when rephrasing is employed are:

1. delays enforced by the middleware for comparison of results
2. delays from using the following mechanisms within transactions:
   - Transaction SAVEPOINTs
   - Transaction ROLLBACKs
   - Execution of SELECT statements after WRITE statements (INSERT, UPDATE, DELETE)
   - Rephrasing

The additional delay introduced by the use of rephrasing is delay 2. We have performed an experimental study to estimate delay 2. Delay 1 would exist also in a diverse setup with or without rephrasing. Studies that have reported measures of other delays which are not specific to rephrasing (such as enforcing 1-copy serialisability) can be found in [18], [6] [8]. There are other factors that can influence the degradation of performance that we have not measured in our experimental setup (e.g. rephrasing delays when more than one rephrasing rule is used etc.). The experiments that we have conducted aim to provide an initial estimate of the delays due to rephrasing. A more thorough performance evaluation should also take into account concurrent execution of transactions. As was also noted by one of the anonymous reviewers, for some concurrency control mechanisms, the increase in transaction execution times due to the use of rephrasing, the probability of conflicts due to concurrency may also increase which may further degrade performance.

The experimental setup consisted of three computers. All three computers ran on Microsoft's Windows 2000 operating system, they had 384 MB RAM, and Intel Pentium 4 1.5GHz processors. One machine hosted the client implementation of the TPC-C benchmark. The other two machines hosted the servers (PostgreSQL 8.0 and Firebird 1.5). We used later releases of the servers than the ones used in our bugs study since these earlier releases do not support SAVEPOINTs and ROLLBACKs within transactions. We have not used any commercial servers in our experiments since the license agreements are very restrictive with regard to publishing performance data.

We ran experiments on both diverse and non-diverse setups. In the diverse experiments we always wait for the slowest server response before we can start the next transaction. Therefore the diverse setups here are always slower (other configurations are possible and we have discussed some of these in [9]).

Figure 2 illustrates the sequence of executions within a transaction for the different non-diverse setups. The grey boxes represent the fault tolerance mechanism used whereas the dotted lines represent the added delay from the use of the respective mechanism. Setup a) is the baseline, against which we will measure the added delays. Setups b), c), and d) measure the delays of using the fault tolerance mechanisms when no failures are observed (i.e. the cost of being cautious)[9]. Setups e) and f), measure the cost of re-execution of a statement[10]. These experiments measure delays for a number of situations:

- re-execution of an unchanged statement as a possible protection against transient failures (caused by the so called "Heisenbugs" [12])
- re-execution of a logically equivalent rephrased statement in case the first one has failed self-evidently (i.e. a crash or other exceptional failures)
- re-execution of a logically equivalent rephrased statement to get additional results for comparison on the middleware to increase the likelihood of failure detection for non-self-evident failures

In our experiments we did not use rephrased statements. Instead, the same statement was executed twice. This is a simplification due to the absence of a proper implementation of rephrasing. In the absence of any other data, we wanted to get an initial estimate of the delays that the various fault tolerance mechanisms will produce with the database servers.

The diverse setups have a similar structure. The only difference is that in diverse setups we only use 1 SAVEPOINT (at the beginning of the transaction) rather than before each write statement and therefore we may also have only one ROLLBACK (at the end of transaction). For setups e) and f), this means that we first execute every statement once then we ROLLBACK to the beginning and execute all the statement again. So the difference between the diverse and non-diverse setups is a different level of granularity of using SAVEPOINTs/ROLLBACKs.

[7] The TPC-C experiments were carried out with 1 emulated client and 1 warehouse with client think times set to 0.

[8] These studies also provide some optimisation procedures for 1-copy serialisability.

[9] b) detection of erroneous writes; c) SAVEPOINT are used before write statements for finer grained recovery; d) both SAVEPOINTs are used and the modified rows are read back (combination of b) and c));

[10] e) optimistic (on writes) rephrasing: each statement is executed twice; to ensure that the state of the database remains unchanged during the second execution of the write statement we use SAVEPOINTs and ROLLBACKs; f) pessimistic rephrasing: same as e) but the written rows are also read to protect against erroneous writes.

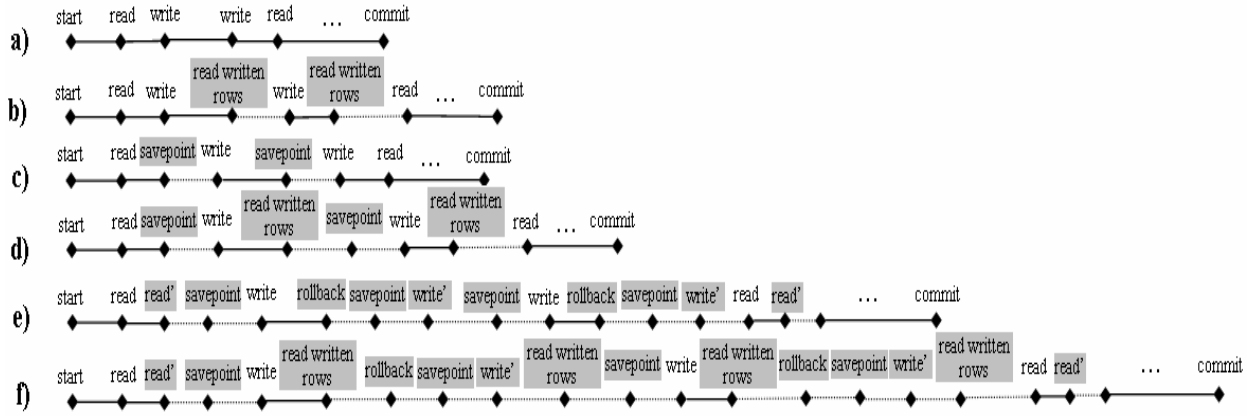


**Fig. 2. A transaction execution sequence in the experimental setups. The shaded boxes represent the fault tolerant mechanism used and the dotted lines represent the additional delays from their use. The second executions of the statements are proxies for rephrased versions of statements.**

The full results of these experiments are given in Table 2. The first column explains the setup under which the experiment was run. The following 4 columns spell out which fault tolerance mechanisms were used (if the cell is blank then the respective mechanism was not used). The following 3 columns show the average execution time of a transaction, and the last 3 columns show the added delay (in percentages) proportional to the baseline of each setup. The first six rows contain the results for each of the setups we explained earlier (and illustrated in Figure 2).

The last two rows are structurally the same as setups (e) and (f) respectively. However in these experiments we have tried to simulated the effect of a simple learning rule: if after 1000 executions a statement has been found to be correct then we stop rephrasing (in our simulation it means we stop executing the statement twice for both setups and additionally stop executing the SELECT statement that read the modifications of the write statements for setup (h)).

The delays seem to be higher proportionally in PostgreSQL than in Firebird. This is because the execution time of COMMITs is smaller in Firebird for the experiments with larger number of SELECT statements. The number of write statements to be COMMITed always remains the same in all experiments (even in the ones with 2 executions of statements, since the first execution of a write statement is always ROLLBACKed). Comparing the setups a) with e) we can see that even though in setup e) every statement is being executed twice the average execution times of the transactions are not simply twice the execution time of transactions in setup a). This is explained by the fact that the number of transactions remains the same (i.e. we still have the same number of COMMITs) and also the data may be stored already in the RAM which reduces the execu-

**Table 2 Performance effects of the various fault-tolerance schemes. Each experiment is run with loads of 10,000 transactions**

| Setup | | | | | Average Transaction Execution time (milliseconds) | | | Delays proportional to the baseline (%) | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Setup description (with reference to Figure 3) | SAVEPOINTs | ROLLBACKs | 2 executions for each statement | SELECT after Write statements | PG 8.0 | FB 1.5 | Diverse PG 8.0 & FB 1.5 | PG 8.0 | FB 1.5 | Diverse PG 8.0 & FB 1.5 |
| Baseline (a) | | | | | 228 | 306 | 343 | | | |
| Detection of erroneous writes (b) | | | | √ | 292 | 356 | 434 | 28.3 | 16.3 | 26.5 |
| Finer granularity of recovery (c) | √ | | | | 240 | 308 | 350 | 5.3 | 0.4 | 1.8 |
| Combination of b and c (d) | √ | | | √ | 305 | 364 | 433 | 33.9 | 18.6 | 26.0 |
| Optimistic (on writes) Rephrasing (e) | √ | √ | √ | | 353 | 450 | 489 | 54.9 | 46.9 | 42.3 |
| Pessimistic Rephrasing (f) | √ | √ | √ | √ | 496 | 601 | 699 | 118. | 96.2 | 105.5 |
| Learning Optimization (g) | √ | √ | √ | | 256 | 325 | 402 | 12.6 | 6.2 | 17.3 |
| Learning Optimization (h) | √ | √ | √ | √ | 278 | 341 | 524 | 22.5 | 11.4 | 52.6 |

tion time of the second statement. The same holds when comparing results of setups b) with f).
Since the numbers in Table 2 represent point estimates (i.e. they are single runs of an experiment per setup) we have repeated the experiments for setup a) and f) to measure the non-deterministic variation that may exist between the different runs. We observed a very small difference (less than 1% for 5 out of six of the experiments and less than 3% for all). Hence we can trust with a higher degree of confidence that the observations documented in table 2 represent closely the 'true' differences between different setups.

## 5. Discussion

We presented in section 2 the architecture we propose for a fault-tolerant server employing rephrasing. The middleware used would make use of a rephrasing algorithm. Any fault-tolerant solution, which makes use of server diversity would need to have "connectors" developed as part of the middleware to translate a client sent statement to the dialect of the respective server. This is because each server 'speaks' its own dialect of SQL. The rephrasing algorithms can also be part of these connectors. A related point is that database servers offer features that are extensions to the SQL standard, and these features may differ between the servers. Therefore for applications which require a richer set of functionality data diversity would be attractive alone as it would for instance allow applications to use the full set of features. A complex statement, which can be directly executed with some servers but not others, may need to be rephrased as a logically equivalent sequence of simpler statements for the latter. For example, the TRUNCATE command is a PostgreSQL specific feature (and is buggy in version 7.0; see bug 20 [17] for details). In its stead the DELETE command can be used to workaround the problem. The DELETE command is also implemented in Firebird and all the other SQL compliant servers.

Since most of these rules are transformations of the SQL grammar, they are amenable to formal analysis. Thus, despite the additional implementation, high reliability can be achieved with a combination of formal analysis and testing of the new code.

The results presented in section 3 demonstrate that a small number of rephrasing rules can help with server diagnosis and state recovery. We observed that a limited set of generic rephrasing rules that we have defined (14 in total) can be used as workarounds for at least 80% of the non-self-evident failures caused by read statements and at least 60 % of failures caused by write or DDL statements in any of the open-source 2-diverse setups in our study. We have also observed that using data diversity without design diversity would lead to a large number of *specific rephrasing rules* to workaround certain failures. Implementing such rules might require a substantial amount of new implementation, which itself may be faulty, thus, reducing the possible reliability gains that can be obtained from their use.

Rephrasing has been proposed as a possibility to detect failures that would otherwise be un-detectable in some replication settings. The possible benefits of this approach could be its relatively low cost in comparison with design diversity, and also that it can be used with or without design diversity allowing for various cost-dependability trade-offs. Possible setups include:

- In non-diverse redundant replication settings, if high dependability assurances are required, the only option available would be to rephrase all the statements sent to the server. This can lead to high performance penalties. To reduce the performance penalty some form of learning strategy can be applied, e.g. keep track of all the statements that have been rephrased. If the rephrased statement keeps giving the same results as the original statement then confidence is gained that the original statement is giving the correct result and the statement does not have to be rephrased in future occurrences (what we did in setups g) and h) of the TPC-C experiments). The other dimension is to stop sending the client-version of the statement to a server if it always gives an incorrect result. In this case the middleware can flag each occurrence of this statement and use the rephrased version of it without sending the original statement to the server [2]. This reduces the time taken to respond to the client.
- In a diverse server configuration a less rephrasing-intensive approach may be used where only the read statements (i.e. SELECTs) that return different results are rephrased (assuming that at least two servers are running in parallel so that a mismatch is detected). The rephrasing is also done for all the write statements (to ensure that the state of the database is not corrupted). Since a smaller set of statements needs to be rephrased the performance is enhanced. The non-self-evident identical failures, however, (we observed 4 of these in the study with known bugs of SQL servers [1]) will not be detected. To further enhance the performance the same learning strategies can be used as in the previous setup.

## 6. Conclusions

We have reported previously [1] on the dependability gains that can potentially be achieved from deploy-

ing a fault-tolerant SQL server, which makes use of diverse off-the-shelf SQL servers. From studying bugs reported for four off-the-shelf servers we reported that failure detection rates in 1-out-of-2 configurations was at least 94% and this increased to 100% in configurations which employed more than two servers. However fault tolerance is more than just failure detection. In this paper we reported on the mechanism of data diversity and its application with SQL servers in aiding with failure diagnosis and state recovery. We have defined 14 generic 'workaround rules' to be implemented in a 'rephrasing' algorithm which when applied to a certain SQL statement will generate logically equivalent statements. We have also argued that since these rules are transformations of the SQL language syntax, they are amenable to formal analysis and dependability gains from employing rephrasing are achievable despite the development of a bespoke new code.

We also outlined a possible architecture of a fault tolerant server employing diverse SQL servers and detailed how the middleware used in it can be extended to also handle rephrasing of SQL statements.

We also presented some performance measurements from experiments we have run with an implementation of the TPC-C benchmark [5], which gave initial estimates of the likely delays due to employing rephrasing.

Further work that is desirable includes:

- demonstrating the feasibility of automatic translation of SQL statements from, say ANSI/ISO SQL syntax to the SQL dialect implemented by the deployed SQL servers. We have completed some preliminary work on implementing translators between MSSQL and Oracle dialects for SELECTs, and between Oracle and PostgreSQL dialects for SELECT, INSERT and DELETE statements;
- developing the necessary components so that users can try out diversity in their own installations, since the main obstacle now is the lack of popular off-the-shelf "middleware" packages for data replication with diverse SQL servers. This would also include implementing a mechanism of maintaining (adding/removing) rephrasing rules as add-on components in the middleware.

## Acknowledgment

This work has been supported in part by the Interdisciplinary Research Collaboration in Dependability (DIRC) project funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC). Authors would like to acknowledge the anonymous reviewers for the thoughtful comments and useful suggestions.

## Bibliography

1. Gashi, I., Popov, P., Strigini, L. *Fault diversity among off-the-shelf SQL database servers* in *DSN'04*, 2004, Florence, Italy, IEEE Computer Society Press p. 389-398.
2. Popov, P., et al. *Software Fault-Tolerance with Off-the-Shelf SQL Servers* in *ICCBSS'04*, 2004, Redondo Beach, CA USA, Springer p. 117-126.
3. Popov, P., et al. *Protective Wrapping of OTS Components* in *4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*, 2001, Toronto
4. Ammann, P.E. and J.C. Knight, *Data Diversity: An Approach to Software Fault Tolerance,* IEEE Transactions on Computers, 1988, C-37(4), p. 418-425.
5. TPC, *TPC Benchmark C, Standard Specification, Version 5.0.* 2002 http://www.tpc.org/tpcc/.
6. Patiño-Martinez, M., Jiménez-Peris, R., Kemme, B., and Alonso, G., *MIDDLE-R: Consistent database replication at the middleware level,* ACM Transactions on Computing Systems, 2005, 23(4), p. 375-423.
7. Bernstein, P.A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. 1987, Reading, Mass.: Addison-Wesley. 370.
8. Jimenez-Peris, R., M. Patino-Martinez, G. Alonso, and B. Kemme, *Are Quorums an Alternative for Data Replication?,* ACM Transactions on Database Systems, 2003, 28(3), p. 257-294.
9. Gashi, I., Popov, P., Stankovic, V., Strigini, L., *On Designing Dependable Services with Diverse Off-The-Shelf SQL Servers*, in *Architecting Dependable Systems II*, R. de Lemos, C. Gacek, A. Romanovsky (Eds). 2004, Springer-Verlag. p. 191-214.
10. EnterpriseDB, *EnterpriseDB*. 2006 http://www.enterprisedb.com/.
11. Janus-Software, *Fyracle*. 2006 http://www.janus-software.com/fb_fyracle.html.
12. Gray, J. *Why do computers stop and what can be done about it?* in *6th International Conference on Reliability and Distributed Databases*, 1987
13. Tso, K.S. and A. and Avizienis. *Community Error Recovery in N-Version Software: A Design Study with Experimentation* in *FTCS-17, Pittsburgh, Pennsylvania, July 6-8*, 1987 p. 127-133.
14. Sutter, H., *SQL/Replication Scope and Requirements document*, in *ISO/IEC JTC 1/SC 32 Data Management and Interchange WG3 Database Languages*. 2000. p. 7
15. Gashi, I., *Rephrasing Rules for SQL servers*. 2006 http://www.csr.city.ac.uk/people/ilir.gashi/Bugs/.
16. Gashi, I., *Tables containing known bug scripts of Firebird 1.0 and PostgreSQL 7.2*. 2005 http://www.csr.city.ac.uk/people/ilir.gashi/Bugs/.
17. Gashi, I., *Tables containing known bug scripts of Interbase, PostgreSQL, Oracle and MSSQL.* 2003 http://www.csr.city.ac.uk/people/ilir.gashi/DSN/.
18. Lin, Y., Kemme, B. et al., *Middleware based Data Replication providing Snapshot Isolation* in *ACM SIGMOD Int. Conf. on Management of Data*, 2005, Baltimore, Maryland, USA, ACM Press p. 419-430.