A Cloud Database with Transactions

Malcolm Crowe, University of the West of Scotland

1. Introduction

With the arrival of cloud computing, many authors have sought to relax consistency requirements in order to maximise the availability and performance of database systems in the cloud (e.g. [1]). This paper represents a contribution in the other direction, maintaining full consistency, but requiring the availability of specific transaction server(s) before accepting a transaction commit.

The difference here is between the availability of the cloud database service (Google Bigtable [2] for example) and the availability of the host(s) for a specific element of the Bigtable. With cloud computing, people are contemplating elastic services where owing to popularity and globalisation, the requirements of a particular data set need to be satisfied by very large numbers of servers. In the background is the famous CAP theorem [3], [4] that there is a conflict between consistency, availability and tolerance to network partitioning, and no system can achieve all three. The theorem was originally stated in the context of web services, but the C of CAP is also the C of ACID, and for databases strong consistency is achieved by requiring ACID transactions. Transaction management imposes “a total order on all operations such that each transaction looks as if it were completed in a single instant” [4]. Weaker forms of consistency are unacceptable from the viewpoint of this paper, but availability is further discussed in section 8 below.

A number of recent database management systems such as ElasTras [5], VoltDB [6], and Xeround [7] have come up with elegant solutions for cloud computing that preserve the ACID properties of transactions, with the help of optimistic concurrency control, and database partitioning. Generally, though, their preoccupation with high throughput has led to restrictions in other directions. VoltDB, for example, only supports transactions at the level of stored procedures, which poses difficulties for a lot of standard OLTP applications, such as holiday booking or book purchasing, where the client wants to be sure a seat or book is available before authorising payment.

The open-source Pyrrho DBMS [8], a relational database with fully ACID transactions and SQL2008 [9] compliance, has been used as a vehicle for exploring multi-server databases for example in developing a semantic web capability, and for these reasons has always used optimistic concurrency control. In its first attempt at approaching cloud computing (v.4.2), it envisaged a single transaction master for each database, and focussed on making the transaction serialisation process as efficient as possible, by separating it on the one hand from query processing and on the other from storage and retrieval operations. In the most recent version it allows databases to be partitioned by the primary keys of tables on a shared-nothing basis, for example, by country. With care this can ensure that many transactions only require to be committed against one partition, achieving a reasonable degree of scalability.

2. Master-slave database architectures

A feature of Pyrrho is the availability of the complete history of the database. This works well for long-term business records, but not very well for the sort of transient data that is created during trend analysis and planning. Of course, such tables can be dropped and thus disappear from normal view, and fresh tables can be created using the same names (names of database objects only need to be currently unique, not historically unique). Nevertheless, if the historical record of the database becomes a source of pride, there can be some irritation that users are polluting this history with their ad-hoc tables.

Accordingly, Pyrrho has a facility to connect to several databases at once. New database objects such as tables created during such a session will be added to the first-named database, while objects in the other databases remain accessible subject to the permissions that have been granted to the user. Connections can be opened and closed for very little cost, since the database file is only fully processed the first time the server accesses it.

For example, a connection list of form “Files=A,B,C,D”, would be appropriate for a connection for performing some sort of data analysis on database D, using tools stored in database C, where database B contains partially-completed analyses, and database A is being used for temporary

results that will be deleted at the end of a session. Database B can be archived and a fresh analysis database constructed for the next period of analysis, while the tools in C are kept under revision for future use, separate from the live database D.

With this scenario, connections of form “Files=D” would be used for normal business operations on D, connections of form “Files=C,D” would be used for adding new tool objects such as stored procedures or views useful for data analysis, and connections of form “Files=B,C,D” would be used to create the analysis tables. B and C would probably not be usable on their own, but in Pyrrho it is perfectly acceptable for a stored procedure in C to reference an object in B.

This approach works well to support the different files in a partitioned database, as we will see in the next section. A connection that only requires access to one of the partitions will typically connect to two database files: the desired partition, and the base database file that contains tables that were not partitioned, and the schema on which the partition is based.

With the use of operating system integrated user identities, the user identity can be expected to be valid in all databases involved in the connection. Authorities are a different matter. Generally at most one of the databases will be modified in any transaction, and the authority string chosen for the connection should be the correct one for such a modification. It is not expected or required that authority identifiers in different databases should match in any way.

There are some limitations on usage in such multi-database connections:

- All data definition, drop, and access control statements affect only the local database.
- Object integrity cannot create dependencies on other databases: generally, schemas must not contain references to other databases. This means, for example, that new referential constraints can only be specified where the referenced table is in the local database. On the other hand, subtypes (including URI-based subtypes) are constructed in the receiving database as needed when data from another database is inserted in the local database.

Data manipulation can affect all databases in the connection. It is relatively unusual for a single transaction in a multi-database connection to result in changes in more than one database (for example in the above scenario with four databases, at most one would be modified in any transaction). The occurrence of a transaction that modifies more than one database makes a permanent link between the databases since for example the transaction cannot be verified unless all the participating databases are online. By default Pyrrho verifies such transactions for consistency each time any of the participating databases is loaded, so that all participating databases must be available to the server (they may of course be remote). A database can be made known to the server through use of configuration files, or using a connection string that refers to all of the databases.

3. Partitioning a Database

If a database becomes too large or too busy for the server(s) on which it is provided, an option is to partition it to create a “distributed database” (Ceri and Pelagatti, 1984). When the partitioning is done horizontally by selecting ranges of values of fields in the primary key (typically a multi-column primary key is used), the result is nowadays called “shared-nothing replication”. From October 2010, Pyrrho supports this architecture. Configuration files are used to specify the relationship between databases, partition files, and the partitioning mechanism.

Since Pyrrho retains a historical record of the database, creation of a new partition is an event in this history. Each partition has its own database file. The complete database consists of a collection of database files, one of which (the “root”) has the same name as the database. All of the files agree on the initial part of the file, and effectively the collection of files form a branching tree-like structure. Each new partition starts out as a copy of a database file in the set, and a configuration file entry is added to specify its file name, the name of the (“base”) file it branches from, and the new partition key data for the table(s) it contains. Pyrrho servers only maintain indexes for the partition they have loaded, and so a cold start is required for the base file server(s). No data is shared between partitions (e.g. lookup tables will generally remain in the root file), apart from information contained in the logs.

When supplying the connection string for a partitioned database, the client must specify all of the database file partitions it wishes to connect to. A client can obtain this set dynamically from the root database using system tables.

A connection that needs to refer only to the data in a single partition only need connect to that partition, but generally a multi-file connection will be required. For example, alteration to a table which references the base partition will require a connection that includes the base partition. Note that many cross-partition operations in the logs will be treated as cross-database transactions, and the server will try to verify coherence of the participating databases.

Schema information should be in the root database, so tables should be initially created there. A new partition can be configured to contain existing data: this data will then no longer be available in the base database. It is possible to add a new empty partitioned table to an existing partitioned database. Any other change to partition information will probably be unusable. Pyrrho only modified database files by appending new transaction data to them. This applies even to transactions that delete records or drop tables. The location of unmodified data is never changed. Bringing a database copy up to date always amounts to adding some additional information at the end of the file.

There is no supported way of recombining partitions. However, it is possible to recover data from partitions by considering how the server will treat the transaction logs. For example, if partition information is lost, the initial data from a partition will reappear in the base database next time the base database server is reloaded, and the file used for a partition will contain the base data information together with the data in its partition. Although the root part of the database must be available, it should not be modified, and thus should not become a bottleneck.

4. System Table for Partition Information

The Sys$Horizontal table has entries only for partitioned databases with a table where only a portion is maintained in the data file, and gives access to configuration information.

Field	DataType	Description
DataFile	Char	The database file containing the partition
Table	Int	A specific table
Column	Int	A specific column for value bounds
Max	Char	Value bounds for the specified column (default unbounded)
MaxInclude	Boolean	Value bounds for the specified column (default true)
Min	Char	Value bounds for the specified column (default unbounded)
MinInclude	Boolean	Value bounds for the specified column (default true)

5. Example Partitioning Configuration

Suppose that one of the tables on a database has so much traffic that it is worthwhile placing a part of it on a different server, and this part can be chosen to minimise the number of transactions that access both parts of the table. We can delegate single-partition transactions to the partition server, while cross partition transactions must be coordinated by the master server, which will be the base server by default.

For definiteness, let us suppose the database name is “Sales”. The configuration file (on hosts H and J) might read (for simplicity we split just one table):

```
<Config>
  <Databases>
    <Database File="Sales" >
      <Server Host="H" />
    </Database>
    <Database Base="Sales" File="Sales_1" DefPos="372">
      <Server Host="J" />
          <Table DefPos="66">
           <Rows>
             <Column DefPos="158" Min="2" Max="2" />
           </Rows>
          </Table>
    </Database>
  </Databases>
</Config>
```

Both hosts will hold the schema information from the common portion of the database (this common schema information cannot be changed). The full index of the specified table will not be on either

server, but all indexes of all other tables are on host H. A transaction limited to the Sales_1 partition should specify Files=Sales_1;Host=J in the connection string (thus directly connecting to J). A transaction needing other partitions should specify them: Files=Sales,Sales_1;Host=H in this case. (For best results the first file specified should be on the Host being connected to.)

Tables and Columns are referred to by DefPos rather than by name, in case they have been renamed since their creation.

The algorithm for horizontal partitioning assumes that a partition holds all rows whose keys match the value bounds specified, except for rows belonging to a child partition.

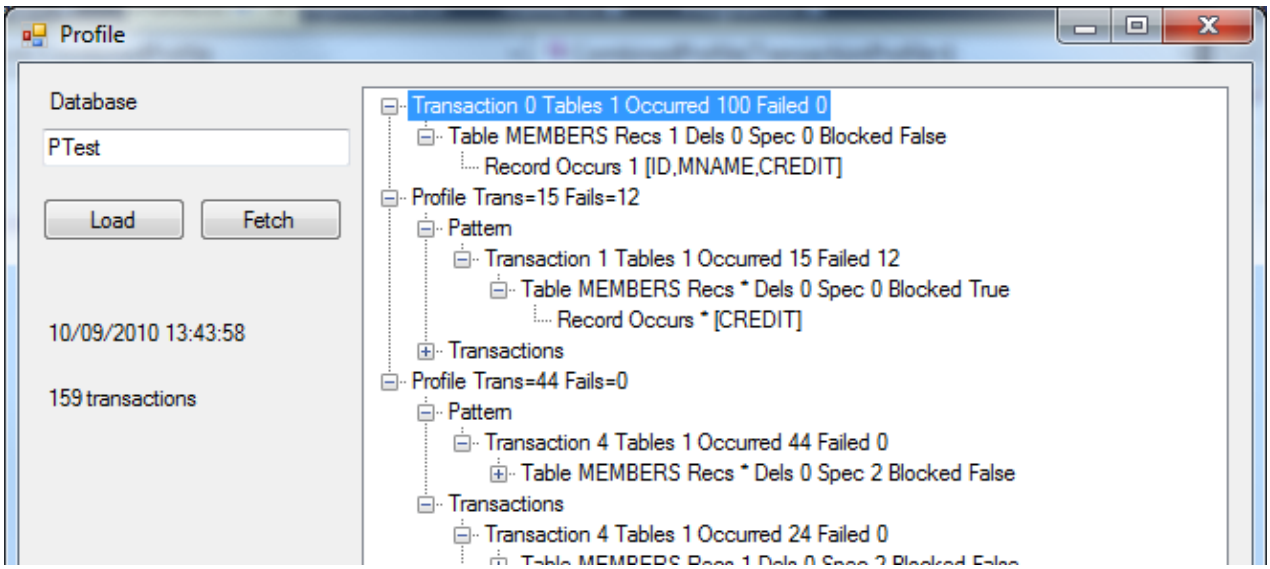
After the first cross-partition transaction, the Sales and Sales_1 files cannot be used separately, as validation of any cross-partition transaction will check their information is consistent. The two files could be configured to run on the same host, but there is no easy way to put them back together.

6. Using Multifile Connections

The purpose of transaction profiling is to examine the relative frequency of different kinds of transactions, and especially to identify the causes of transaction conflicts. Good database and application design will seek to minimise transaction conflicts for normal operation of the business processes, but in general some transaction conflicts are inevitable.

With Pyrrho, transaction profiling is something that is enabled for a period to examine the above issues. Transaction profiles are persisted not in the database itself, but in XML files: this is because it is a record not of the entire database activity, but just the periods for which profiling is enabled. Profiles can be deleted without harming the database in any way.

There is a convenience utility called ProfileViewer which displays the profile in a readable tree-view format. The profile can either be “fetched” from the server (assuming profiling is enabled), or “loaded” from the XML file (in which case ProfileViewer expects to find the xml file in its working folder).



When profiling is turned off or on for a database called *name* profiling information is destructively saved as or if available loaded from an XML document with name *name.*xml. Thus a database administrator can carefully take a database offline by throttling, and then turning off profiling to record a snapshot before shutting down a server, and in this way a full profile of normal operations can be maintained. This level of completeness for profile information will not be achieved if the database server is simply killed.

If profiling is enabled, any failed transaction will report its profile. The system profile table will contain the number of successful and failed transactions recorded for this profile: the number of successful transactions will be based on the entire history of the database, while the number of failed transactions recorded will be based on the available information from recorded periods of full profiling (or since the time profiling was enabled for the server).

If profiling is turned on, the tables described in this section enable inspection of the real-time state of the profile information, always excluding any information about transactions in progress. The profile

viewer described in section 4.6 obtains profile information from these tables or from the XML document, and also groups profiles with similar pattern (for example where everything is the same apart from the number of affected rows).

The Profile$ system table records the transaction profiles for the database.

Field	DataType	Description
Id	Int	The transaction profile identity
Occurrences	Int	The number of times this profile has occurred
Fails	Int	The number of failures recorded for this profile
Schema	Boolean	Whether this transaction includes schema changes

Further details for this profile are contained in the following tables.

The Profile$ReadConstraint system table records the read constraint for a transaction profile.

Field	DataType	Description
Id	Int	The transaction profile identity
Table	String	The current name of the table
ColPos	Int	The defining position of a read column whose update is blocked
ReadCol	String	The current name of a read column whose update is blocked

The Profile$Record system links a transaction profile with a record profile.

Field	DataType	Description
Id	Int	The transaction profile identity
Table	String	The current name of the table
Rid	Int	The record profile identity
Recs	Int	The number of records altered with this profile

The Profile$RecordColumn table records the columns containing added or updated data in a record profile.

Field	DataType	Description
Id	Int	The transaction profile identity
Table	String	The current name of the table
Rid	Int	The record profile identity
ColPos	Int	The defining position of an affected column
RecCol	String	The current name of an affected column

The Profile$Table table records the profile of delete operations for a specific table as well as providing information about update blocking.

Field	DataType	Description
Id	Int	The transaction profile identity
Table	String	The current name of the table
BlockAny	Boolean	This profile blocks on any concurrent update of the table
Dels	Int	The number of deletions in a transaction
Index	Int	The defining position of an index with specific records
Pos	Int	The defining position of the table
ReadRecs	Int	The number of specific records whose update is blocked
Schema	Boolean	Whether the profile changes the table schema

If BlockAny is true, Index and ReadRecs will be 0; and if there are Profile$ReadColumn entries blocking is limited to these columns.

7. Pyrrho and Cloud computing

By default a Pyrrho server uses local memory and local durable storage to provide access to a set of databases for which it acts as transaction master.

Pyrrho servers can optionally be configured to operate as a cloud computing service, with servers operating at Storage, Master and DBMS levels. Full details of the configuration file are given in section 10. The principles adopted are that a DBMS can provide direct or indirect access to a number

of databases; each database must have a single transaction master, and cloud storage for a database can be replicated, but can be updated only by the transaction master.

As a consequence, a Storage server needs to have a transaction Master or DBMS configured for each database that it stores. If a transaction master is configured for a database, then it can have a number of DBMS's configured for direct access, and further DBMS's can be added by the transaction master. These DBMS's can be used for direct or indirect access to the database, and optionally a DBMS can maintain a local copy of the database enabling read access when the transaction master is unavailable.

If the configuration file for the storage server specifies a DBMS for a database without specifying a transaction master, then the DBMS takes the role of transaction master for its own operations, and can also be used for indirect access. Thus a transaction master for a database may itself be a DBMS, in which case it can do query processing, SQL, SPARQL etc on behalf of clients (Level 4 in the Pyrrho protocol, see section 8.5). Otherwise, its role is to arbitrate and commit transactions sent to it by DBMS's. Unlike a server playing the DBMS role, it needs to keep track only of authorised servers and user-defined data types for each database it masters. This saving in complexity makes means that a single transaction manager can easily handle traffic from multiple DBMSs.

The concept of indirect access to a DBMS is discussed further in chapter 11 (remote transaction master). It allows implementation of large databases whose data is partitioned over several servers.

8. Configuring servers and availability

Pyrrho can be operated on laptops, PDAs and phones that have the .NET framework. By default all databases are held on the local machine or device. However, such a local server can be configured to access master copies of the data held on a remote Pyrrho server. This results in a form of cloud computing. Multi-database connections can include both local and remote data, which can be combined in query processing. Such combinations can be completely ad-hoc, so they are allowed to be specified in the connection string (they do not require server configuration). Servers can also be configured to work with remote databases, by using a configuration file.

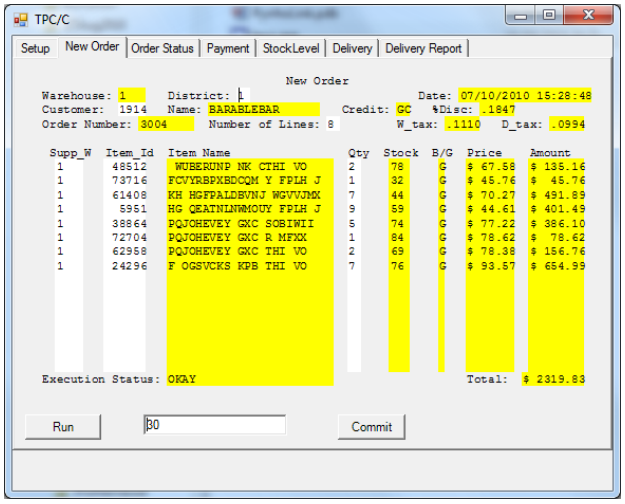
The synchronised storage mode can also be used if the database is small enough to fit comfortably in memory. If this is not desirable, with the Remote storage mode a local server can simply keep track of schema information for the remote (cloud) database(s) it is allowed to access, and search and manipulate their data using ordinary SQL. In addition, by using multi-database connections, a local server can efficiently integrate data from local and remote databases, for example by using functional dependency and constraint information. Query processing discovers minimal rowsets to obtain from the cloud database(s) for combination with rowsets from the local database(s).

For example, a company agent could maintain a database of visits and potential sales on a PDA, and remotely access the corporate database to obtain customer and product information. An application on the PDA could combine the two sets of information in useful ways.

If the remote database is inaccessible, a cached copy of a valid state of the database may still be available. Pyrrho allows such data to be readable, and a transaction can even be started. But the transaction cannot be committed without recourse to the transaction master, and will then fail if a transaction has modified any data accessed by the client since the time the copy was last validated. If the client does not seek to make changes, then it is as if the client's work took place at an earlier time (the time the copy was made). This is relevant to the discussion of partial synchronisation in [4]: if messages are lost, however, Pyrrho will report that the transaction has failed.

9. Transaction Rate and Benchmarking

The TPCC benchmark [10] is cited in many of the papers referred to, and has been tried in a variety of configurations on the database system described above. The TPCC is a reasonable model of classic online transaction processing, although we outline below two major difficulties with this benchmark for cloud computing.

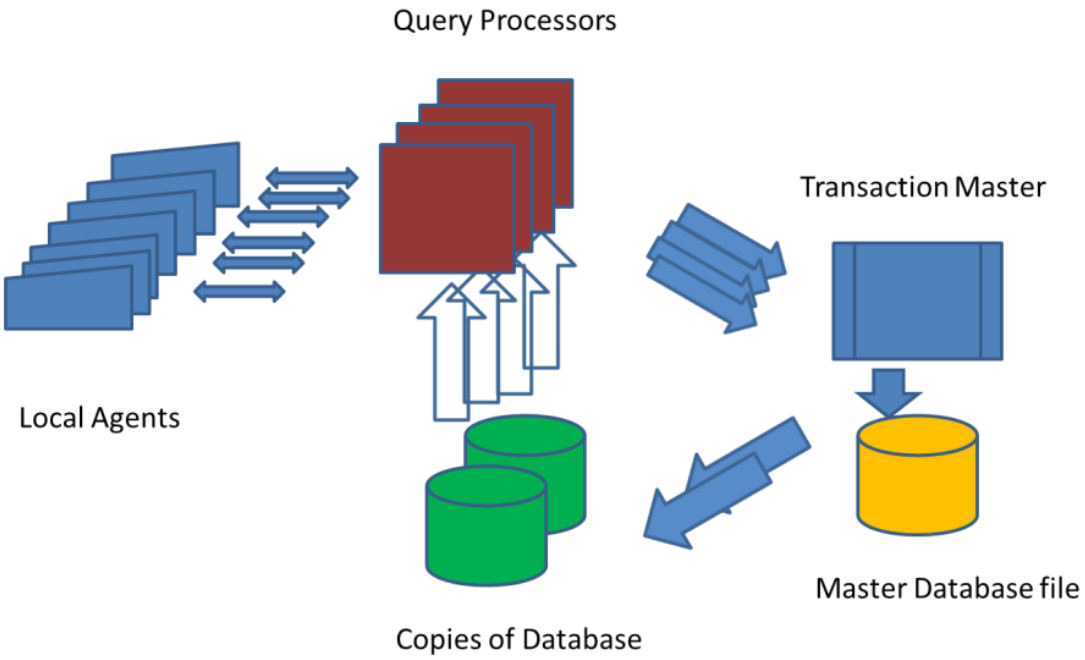


The above illustration highlights in yellow data that is retrieved from the server during the new_order transaction: the entries highlighted in white originate at the client. With a PC acting as query processor, a single client at 33 transactions per second uses 50 MB/s of network bandwidth, even though the transaction traffic is only 55KB/s written to permanent storage. For example, the amount of stock requested for each item is always calculated from the current stock level for that item, so multiple retrieval operations take place in each new order transaction. With multiple agents, the TPCC benchmark design quickly saturates the network. (Of course, no human operator could generate even one transaction per second.)

Secondly, the new order is required to operate as a single transaction that updates a field called d_next_o_id which is held per district. This is incremented by the new_order transaction, so concurrent new_order transactions for the same district are very likely to conflict. At first sight, this per-district behaviour makes partitioning the warehouse by district look attractive. Unfortunately, this partitioning scheme does not work well because every new order updates the stock levels of a random set of warehouse items. To me this problems seems to impose a hard limit on the maximum transaction speed N: if a query processor takes 0.03 sec to prepare a transaction, then N/33 transactions will have committed in the meantime, so with 10 districts there are about N/300 conflicting transactions. To me this seems to suggest N<600 tps.

Nevertheless it is interesting to explore the other ideas in this paper as they could apply to the TPCC benchmark. Imagine that the d_next_o_id problem is solved. The network loading from agent to query processor of 50MB/s suggests a reasonable level of 3 agents per query processor; that is, each query processor can manage 100 new-order transactions per second. 10 query processors could share a disk (or storage server). 20 such clusters could be managed by a single transaction master for a total transaction traffic of 120MB/s. The result of committing transactions would be that the database (log) file would grow at 90 MB/s (2000 new orders becomes 9MB for the PyrrhoDBMS).On a Windows 7 PC, 500MB/s is the practical limit for the disk subsystem, so this volume would give the transaction master adequate time to replicate the new data onto the storage servers. Compared with the published record-breakers for TPCC these figures may seem modest.

Some other test results have been more interesting. The VoltDB community claims that using a single-threaded server with a single transaction queue is more efficient than using multi-threading with many channels. However, in my experiments, a single-threaded simple server with 100 clients accessing the request socket can only manage 50 requests per second, while a multi-threaded server with 100 open connections can do 55000 requests per second. This experiment helped with the TPCC sketch described above.



10. Conclusions

This paper has outlined a scalable approach to database provision that meets many of the requirements of cloud computing, while retaining SQL2008 compliance and fully ACID transactions. It is now one of a number of approaches that use optimistic concurrency control to solve many of the issues that are important for cloud computing while refusing to accept any departure from consistency or transaction integrity. All have come up with ways of maximising transaction efficiency and availability for the applications they consider subject to this overriding constraint.

It is possible to imagine an environment in which all of these solutions would work together on top of a common storage model. This might be a fruitful field for further research. It is a pleasure to thank Tim Lessner for reading a draft of this paper.

References

1. Kossmann, D.; Kraska, T.; Loesing, S. (2010): An Evaluation of Alternative Architectures for Transaction Processing in the Cloud, SIGMOD'10, Indianapolis, USA (ACM) ISBN 978-1-4503-0032
2. Chang, F.; Dean, J.; Ghamawat, S.; Hsieh, W. C.; Wallach, D. A.; Burrows, M.; Chandra, T.; Fikes, A.; Gruber, R. (2008): Bigtable: A Distributed Storage System for Structured Data, ACM Transactions on Computer Systems, **26**.2, article no. 4.
3. Brewer, E. (2000): Toward Robust Distributed Systems, 19th ACM Symposium on Principles of Distributed Computing, Portland, Oregon.
4. Gilbert, S., Lynch, N. (2002): Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services, ACM SigACT News, **33**, p. 51-59.
5. Das, S.; Agarwal, S.; Agrawal, D.; Abbadi, A: (2009): ElasTraS: An Elastic, Scalable, and Self Managing Transactional Database for the Cloud. UCSB Computer Science Technical Report 2010-04.
6. VoltDB, Inc (2010): Scalable, Open Source Sql Dbms With Acid. http://voltdb.com
7. Xeround, Inc (2010): Database Scalability and Availability in the Cloud, Technical White Paper, http://www.xeround.com
8. Crowe, M. K. (2010): The Pyrrho DBMS, version 4.3, University of the West of Scotland, UK http://www.pyrrhodb.com
9. SQL2008 (2008): Information Technology – Database Languages – SQL, International Styndard ISO/IEC 9075-1:2008.
10. TPCC banchmark, Transaction Processing Performance Council. TPCC version 5. http://www.tpc.org