Asynchronous Adaptive Delay Tolerant Index Cache Using In-memory Delta Cell

Kun Ma[1,2] and Bo Yang[1]
[1] Shandong Provincial Key Laboratory of Network Based Intelligent Computing,
University of Jinan, Jinan 250022, China
[2] Shandong Provincial Key Laboratory of Software Engineering,
Shandong University, Jinan 250100, China
E-mail: ise_mak@ujn.edu.cn http://kunma.net

Keywords: index, cache, delta cell, query cache, indexing, relational database

Received: October 8, 2015

Relational database indexes, used to speed up access to data stored in a database, are maintained when data in the source table of the index is modified. Therefore, relational database index management can involve time consuming manual analysis and specialized development efforts, and impose organizational overhead and database usage costs, especially in the context of big data. To address this limitation, this paper proposes an asynchronous adaptive delay tolerant index cache using in-memory delta cell. The contributions of index cache are adaptive management and fine-grained delta cell. Finally, our experimental evaluation shows that this simple index cache has the features such as update efficiency with frequent changes, transparency to developers, and low impact on database performance.

Povzetek: Predstavljena je nova oblika indeksa, ki omogoča boljše delovanje relacijskih baz.

1 Introduction

1.1 Background

Relational databases are organized collections of data using schemas such as tables, records and columns. Information retrieval can be made more efficient by using relational indexes to provide rapid access to data stored in a table [1]. An index is a data structure that is created using one or more columns of a base table using balanced trees, B+ trees, and hashes techniques. Indexes are updated when data in the source table of the index is modified. Therefore, indexes maintenance [2] is performed to provide accurate responses to applications that retrieve data in the presence of frequent changes. Generally, an index is updated immediately when data in its source table is modified [2]. Changes to base tables result from statements to insert, update, or delete records in the base table. Maintaining an index immediately may be inefficient due to frequent changes. For instance, a particular record may be modified several times before it is read when evaluating a query. In this situation, only the latest change to this record before the query is concerned. In addition, index maintenance may occur at peak operating times of the database, especially in the context of big data. Thus, the processing power of the database may be drained due to index maintenance operations. And index maintenance has become the bottleneck of big data access.

1.2 Data access with frequent changes

To address the issue of rapid data access with frequent changes, many approaches and strategies have been proposed. The first solution is distributed cache. A distributed cache may span multiple rapid storage nodes so that it can grow in size and in transactional capacity. It is mainly used to store frequently accessed data residing in database and web session data. This solution is popular due to the cheap hardware such as memory, solid state disk, and disk array. In addition, a distributed cache works well on lower cost machines. Ehcache and Memcached are distributed cache for general purpose caching [3], originally intended for use in speeding up data access by alleviating the database load. They feature memory and disk stores, replicate by copy and invalidate, and cache loaders. However, distributed cache might be suited for the scenario in which the data is read frequently. While in the presence of frequent changes, swapped in and out lead to excessive spending on consistency.

The second solution is cache table. Cache table [4] enables persistent caching of the full or partial contents of the relational table in the distributed environment. The content of a cache table is dynamic, which is either defined in advance at setup time or determined on demand at query time. Although this solution exploits the characteristics of short transactions and simple equality predicates, too massive maintenance of cache table and extra storage spaces are needed in the context of frequent changes.

The third solution is caching query results. TxCache [5] is a transparent caching framework that supports transactions with snapshot isolation. It is designed to cache query results, and extends them to produce invalidation tags in the presence of updates. This works when the workload of an application consists of simple exact-match selection predicates. CacheGenie [6] provides high-level caching abstractions for common query patterns, and generates SQL queries and object instances stored in the cache. It can perform this for a subset of query patterns generated by the ORM. These frameworks are suitable with minor changes of data.

The fourth solution is augmented cache. Cache augmented systems [7] [8] enhance the velocity of simple operations that read and write a small amount of data from big data, which are most suitable for those applications with workloads that exhibit a high read to write ratio. Some query intensive applications augment a database with a middle-tier cache to enhance the performance. In the presence of updates to the normalized tables, invalidation based consistency techniques delete the impacted key-value pairs residing in the cache. A subsequent reference for these key-value pairs observes a cache miss and re-computes the new values. It is difficult to keep consistency in the presence of frequent changes.

The last solution is augmented index. This method improves the traditional index in the presence of updates. Service indexes [9] are created to assist main indexes to record the changes in the presence of updates. They are maintained when there is data manipulation on main indexes. Asynchronous index [10] is a delay index to maintain database indexes or sub-indexes. After the database receives a data manipulation statement to modify particular data, the index associated with this operation is maintained asynchronously until an index maintenance event. In this situation, there are inconsistencies between the delayed index data and actual data. Index maintenance includes delta tables as well as control tables. The challenges of augmented index is how to implement adaptive index management and reduce the cost of maintaining indexes.

1.3 Contributions

The biggest disadvantage of the above five solutions is the bottleneck in the presence of frequent changes. To address this limitation, we attempt to benefit indexes from cache techniques. We call this index cache. Compared with index techniques, we attempt to address index maintenance issue using cache techniques. Unlike cache, index cache is used to speed up read and write at the same time. Thus, index cache is suitable for both high read to write ratio and high write to read ratio. Innovation points of this article lies on the following. First, we provide dynamic management of index cache. Several index management metrics (column access frequency, index maintenance frequency, and deadlock frequency) are collected to compare with the thresholds to determine management actions, such as reorganizing indexes, creating indexes and removing indexes. Proposed actions may be subject to final authorization or may be implemented automatically after the metric threshold values are satisfied. On one hand, the profiler we proposed is general to monitor data query and manipulation statements using JDBC or other middleware. On the other hand, frequency is a corrected metrics. Second, we provide delay tolerant index cache using delta cell. Index maintenance caused by data manipulation associated with this index is delayed within the tolerance. This method is based on an isolation level of a transaction including a query that triggered the index maintenance. In this solution, fine-grained delta cells are used to describe the changes of data. Reset, read, write, and consistency of index cache are also concerned. On one hand, fine-grained delta cells save more storage than delta tables using versioning management. On the other hand, the write of index cache is oriented to cache itself using eventually consistency strategy.

The remainder of this paper is organized as follows. Relevant recent work on dynamic management of index and augmented index is reviewed in Section 2. In Section 3, a description of asynchronous adaptive delay tolerant index cache is presented. First, adaptive dynamic management of index cache is provided to reorganize, create, and remove indexes by the collected metrics. Furthermore, delay tolerant method is proposed to reset, read, write of the index cache to implement the consistency using fine-grained delta cells. Section 4 presents the experimental evaluation of this asynchronous adaptive delay tolerant index cache to illustrate its update efficiency with frequent changes. Brief conclusions and future research directions are outlined in the last section.

2 Related work

2.1 Dynamic management of index

Generally, indexes are created by administrators to speed up data access. In the context of applications with high read to write ratio, indexes are competent to organize data records. Most relational database can provide benefits by controlling index fragmentation and inserting/removing indexes based on database queries [1]. Unfortunately, it involves time consuming manual analysis and specialized development efforts. In some situations, such index management may be performed without an integral management, leading to problems such as the following [11]. First, running query profilers to trace query patterns may cause significant performance overhead on databases. Second, resolving index related issues may impose organizational overhead and slow turnaround time.

Recent researches focus on automatical integral index management for a relational database. For example, dynamic integral index management actions and index management metric thresholds are provided/rectified by administrator. An index metrics collection module automatically collects metric values to determine whether to reorganize or insert/remove indexes [11]. Another case is index monitoring system for selectively maintaining an index [12]. An indication of an index usage criterion associated with each of two or more indexes is provided to efficiently determine exactly what and how indexes are used, and whether the index should be removed and created. Some well-known tuning advisors [13], such as Oracle and Microsoft SQL Server, provide index recommendations for a given work load of queries. Other relational database products also have a separate component that would read a given set of queries and provide the indexing recommendations based on storage, partitioning, and other considerations [14]. Many such products have significant limitations. For instance, the tools are manually controlled. A set of user queries to be analyzed must be captured from production database servers using profilers that can add significant performance. Implementing the index recommendations on the production databases may require IT release cycles, which is often time consuming. Sometimes the metrics are not correct enough to conclude good guiding significance.

2.2 Augmented index

Augmented index is a method to enable indexes to implement the maintenance in the presence of updates. Compared with augmented cache solution, this method is efficient in the context of frequent changes with high write to read ratios. At least a service index [9] is proposed to record the changes caused by main indexes. This is a delayed update method of index maintenance. After data manipulation on main indexes, changes are immediately saved to at least a service index. Maintenance to main indexes is delayed with the help of service index. There are several insufficiencies. First, single table with no more than one index will lead to generate more service indexes. Second, the performance impact on index maintenance is inevitable because main index maintenance is just delayed to update. In the presence of frequent changes, it will also become the bottleneck of data access.

Another augmented index is asynchronous index [10]. Asynchronous indexes may need to be maintained when records of the base table with the index are changed in response to a data manipulation statement. Asynchronously updating an index may improve the efficiency of index maintenance by reducing the number of inputs/outputs needed for index maintenance. This method is particularly efficient for a database table having frequent writes, but infrequent reads. Insufficiencies of this method lies on the following. First, delta tables to store the changes caused by index will occupies huge amounts of required storage spaces. A changes record is stored no matter how many columns have been changed. That indicates that the unchanged column is also stored as long as the record including this column is changed. Second, the merging strategies of massive records in delta tables are not discussed. Without a reasonable merging strategy, the records of delta tables grow fast in the presence of frequent updates.

3 Asynchronous dynamic delay tolerant index cache

3.1 Dynamic management by metrics

We provide a profiler on the read and write statements to regularly monitor the workload (a set of data query and manipulation statements that execute against a database) and control indexes management actions appropriately, to remove unused indexes, to re-organize used indexes, and to create required indexes based on the frequency. We define column access frequency, index maintenance frequency, and deadlock frequency to determine whether to maintain indexes dynamically.

We want to create indexes on frequently accessed column, to remove indexes on frequently index maintenance column, and to re-organize indexes on tables with many deadlocks. The frequency is the broad frequency belonging to one column. We take different frequencies as the metrics of index management.

3.1.1 Column access frequency

Column access frequency reflects the frequency that one column is accessed. When the data in this column are accessed by querying, the column access frequency count is plus an increment. The reason why it is called broad is that the increment is not simply one, depending on the product of the priority weight of this column and the correction factor. When the column access frequency exceeds the threshold, the index on this column should be created to speed up the data access. Column access frequency count f is computed by the query predicates: selection predicates, aggregate predicates, and ordered predicates. To describe the rectification of the column access frequency, we assume the following terminology for a SQL query:

SELECT *target list* **FROM** *table list*
WHERE *qualification list*
ORDER BY *ordered list*

We consider the column access frequency on three predicates: selection predicates (target, exact-match and range selection), aggregate predicates, and ordered predicates. At the beginning, the column access frequency count is zero. Other weights and factors are empirically determined.

We describe the affection of the target and exact-match predicates in turn. Consider the following query with a quantification list consisting of exact-match selection predicates:

SELECT $a_1, a_2, ..., a_n$ **FROM** *table list*
WHERE $a_1 = C_1\ and/or\ a_2 = C_2\ ...\ and/or\ a_m = C_m$

The proposed profiler constructs the rectification of the column access frequency count. If the column is located in the target list, the access frequency count f_{ai} of column ai is plus to the product of the weight ws_{ai} of the column and selection correction factor ks, denoted as $f = f + ws_{ai} * ks$. If the column is located in the exact-match list, the access frequency count f_{ai} of column ai is plus to the product of the weight wm_{ai} of the column and exact-match correction factor km, denoted as $f = f + wm_{ai} * km$.

We describe the affection of the range selection predicates. Consider the following query with a qualification list consisting of range selection predicates:

SELECT *target list* **FROM** *table list*
WHERE $(a_1 > C_1 and a_1 < C_2)$ and/or ... and/or $(a_m > C_{2k}$ and $a_1 < C_{2k+1})$

If the column is located in the range list, the access frequency count f_{ai} of column ai is plus to two times the product of the weight wr_{ai} of the column and range correction factor kr, denoted as $f = f + 2 * wr_{ai} * kr$.

We describe the affection of the aggregate selection predicates. Consider the following query with a quantification list consisting of aggregate selection predicates:

SELECT $function_1(a_1), ..., function_m(a_m)$
FROM *table list*
WHERE *quantification list*

If the column is located in the aggregate list, the access frequency count f_{ai} of column ai is plus to the product of the weight wa_{ai} of the column and aggregate correction factor ka, denoted as $f = f + wa_{ai} * ka$.

We describe the affection of the ordered selection predicates. Consider the following query with a quantification list consisting of ordered selection predicates:

SELECT *target list* **FROM** *table list*
WHERE *quantification list*
ORDER BY a_1 $asc/desc, ..., a_m$ $asc/desc$

If the column is located in the ordered list, the access frequency count f_{ai} of column ai is plus to the product of the weight wo_{ai} of the column and ordered correction factor ko, denoted as $f = f + wo_{ai} * ko$.

3.1.2 Index maintenance frequency

Index maintenance frequency reflects the frequency that indexes are maintained due to data manipulation. When the index is rebuilt by the changed column, the index maintenance frequency of this column is plus an increment. The reason why it is called broad is that the increment is not simply one, depending on the product of the priority weight of this column and the correction factor. Index maintenance frequency count g of one column i is denoted as $g = g + w_{bi} * k$, where w_{bi} is the weight, and k is index maintenance correction factor. When the index maintenance frequency exceeds the threshold, the index on this column should be removed to speed up the data access with changes.

3.1.3 Deadlock frequency

Deadlock frequency reflects the times that one table is locked by the index maintenance. Table access is locked when the index maintenance is not complete. We provide lock frequency h to record the deadlock times. When table access is locked, the deadlock frequency is plus one. When the deadlock frequency exceeds the threshold, a manual check is needed to re-organize the indexes.

3.2 Delay tolerant index cache using delta cell

3.2.1 Delta cell

In order to define the architecture and management actions of the delay tolerant index cache, a mathematical representation of the fine-grained model is necessary. In our solution, we split the storage structure of the basic element of index cache into sets of delta cells divided by column. Delta cell is a fine-grained model of frequent changes of a relational database.

First, we define the elements of a delta cell.

- key is the key of a delta cell. It corresponds to the key of relational changed record before it is divided into cells. key is denoted as 2-tuple $key :< keyname, keyvalue >$, where $keyname$ is the key name of the record, and $keyvalue$ is the key value of the record;
- C is key/value of this delta cell. It is denoted as 2-tuple $C :< name, value >$, where $name$ and $value$ are the name and value of this delta cell respectively.
- $V, V \in \mathbb{Z}^+$, is the version number of this delta cell. It is a non-negative integer. The initial version number of a delta cell is one. When the delta cell is removed, the version number is set zero.

Second, we give the definition of a delta cell. The delta cell is a 3-tuple $< key, C, V >$, where key is the key of a delta cell, C is key/value of this delta cell, and V is the version number of this delta cell. We take schema-free key/value stores to save delta cells. The query of delta cells is through SQL-like HiveQL [15].

As mentioned, delta cells are the first-class artifacts to represent frequent changes. These models are typically created and modified by the profiler we design. One of the techniques used to support index cache management activities is version control. Version control is used during delta cell evolution to keep track of different versions of delta cell artifacts produced over time. Version control enables simultaneous transactions to access the delta cells that stores different versions of the data. When a transaction updates the delta cell, it maintains its previous versions. After index cache is reset, all the versions of delta cells are emptied.

3.2.2 Architecture of index cache

Figure 1 shows the architecture of our proposed asynchronous adaptive delay tolerant index cache. Index cache is the supplement of actual data with indexes. When there are data query statements, the query results are from the merging of index cache and actual data with indexes. When

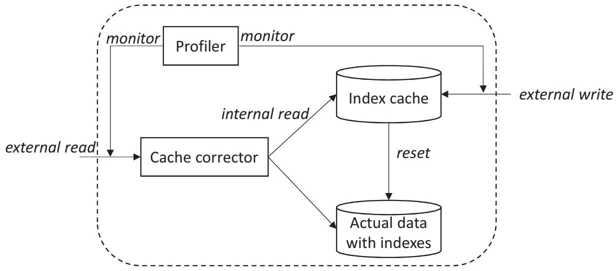

Figure 1: Architecture of asynchronous adaptive delay tolerant index cache.

there are data manipulation statements, all the updates are written to index cache. Index cache is reset triggered by forced update event or idle update event. With this architecture, there is no immediately index maintenance until a forced or idle update event generates. Besides, profiler is used to monitor the external read and write operations to collect the frequencies to adaptively manage indexes.

3.2.3 Reset of index cache

Triggered by a forced or idle update event, the data in index cache is forced to be written to actual data with indexes. When the version number of the delta cell in the index cache is zero, the corresponding record in the actual data with indexes should be removed. When the version number of the delta cell in the index cache is a positive integer, the latest version of this delta cell in the index cache should replace the original record in the actual data with indexes.

3.2.4 Read of index cache

In the architecture of delay tolerant index cache, the query results are from both index cache and actual data with indexes. In order to make the index cache transparent to the user, the query results should be corrected by the inner result corrector in the index cache. When there is a data query statement, the result corrector delivers the query request to both the actual data with indexes and index cache at the same time. In the index cache, only delta cells with a positive integer are to execute the query statement. The query of delta cells is using HiveQL [15]. Afterwards, the results from both index cache and actual data with indexes are merged together. The merging action needs to meet the mergence rules shown below:

- Results with the same key: the query results from index cache replace the results from actual data with indexes.

- Results with different keys: the final results are the union set of both index cache and actual data with indexes.

3.2.5 Write of index cache

With index cache architecture, the write of index cache acts on only itself rather than actual data with indexes. For the creation data manipulation statement, the new record is broken down into several newborn delta cells with version number 1. For the delete data manipulation statement, the removed record is broken down into several destroyed delta cells with version number 0. For the update data manipulation statement, it is divided into two cases. When the changed data is in the index cache, the only thing to do is to update the existing delta cell with the changed data. When the changed data is not in the index cache, the only thing to do is to create a newborn delta cell with version number 1.

In the process of write of index cache, the delta cells are created and updated in order. That is to say that the same delta cell might be updated more than once in a short time. For instance, the data is first inserted, then updated, and deleted at last. Therefore, update merging method is introduced to merge the intermediate result. Afterwards, the delta cells are in no particular order.

Table 1: Merging result of both actions.

Action 1	Action 2	Merging result
Insert	Insert	×
Insert	Update	Insert
Insert	Delete	Ignore
Update	Insert	×
Update	Update	Update
Update	Delete	Delete
Delete	Insert	Insert
Delete	Update	×
Delete	Delete	×

Merging of the delta cells reduces the times of several updates to the final update when data are updated more than once. The mergence rules are shown in Table 1. After continuous two actions of the same delta cell, the final merging result is shown in the third column. The expected merging results might be $impossible$ ($\times$), unchanged ($ignore$).

4 Experiments

We have conducted a set of experiments to evaluate the efficiency and effectiveness of our proposed asynchronous adaptive delay tolerant index cache using delta cell. After a description of the experimental setup, we evaluate three solutions (database without any external index optimizations, augmented index, and index cache).

4.1 Experiment setup

We deploy the experiment architecture with Intel Core(R) i5-2300 @2.80 GHz CPU, 16GB memory. It runs a 64-bit

CentOS Linux OS with a Java 1.6 64-bit server JVM. We use MySQL server 5.6 GA as the relational database. We initialize $1,000,000$ relational records with 20 columns, 1 primary key, and 4 indexes (each index is on one column).

4.2 Update time with frequent changes

We evaluate three different solutions under three circumstances. The x axis is transactional workload (presented using transactions per second), and the y axis is the average time executing $1,000$ data manipulation statements. To increase comparability of the results, $1,000$ statements include one third of new records, one third of changed records, and one third of deleted records. We take asynchronous index as an example of augmented index.

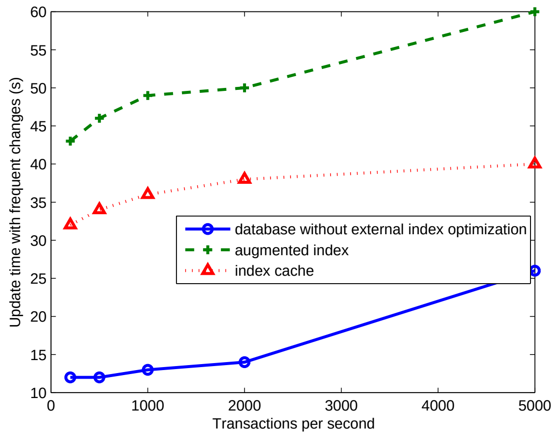

Figure 2: Update time when randomly updating non-index columns.

The first circumstance is randomly updating non-index columns (other 16 columns except 4 index columns). Figure 2 shows the average update time with different transactions per second. Since the frequent changes are not in the index columns, database without external index optimizations solution has the smallest update time with the increasing of transactions per second. Unfortunately, the augmented index works not well due to massive index maintenance. The update time of our proposed index cache is in the middle, because the write of index cache is just in the index cache itself without index maintenance.

The second circumstance is randomly updating 4 index columns. Figure 3 shows the average update time with different transactions per second. Since the frequent changes lie in the index columns, index maintenance issue become the bottleneck of the updates. Database without external index optimizations solution is the worst. When the transactions per second are below 500, the update time of our index cache solution is a little larger than asynchronous index solution. That is due to the reset of index cache in the presence of low frequency. When the transactions per second exceed 500, our index cache is starting to change for

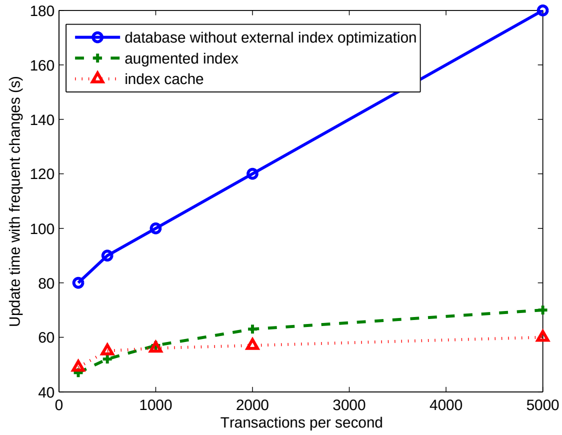

Figure 3: Update time when randomly updating 4 index columns.

the better. That is caused by little reset of index cache in the presence of high frequency.

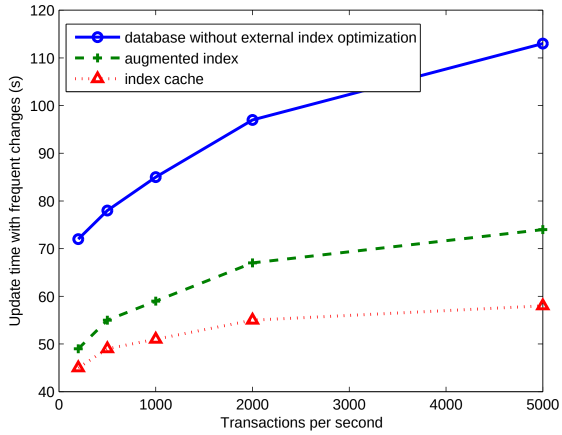

Figure 4: Update time when randomly updating 8 columns.

The third circumstance is randomly updating 8 columns (3 index columns and 5 non-index columns). Figure 4 shows the average update time with different transactions per second. Database without external index optimizations solution is the worst, because index maintenance on 3 index columns takes up the update time. Our index cache works better due to the dynamic index management by metrics. After the experiment, our index cache solution removes 1 index on the frequent updated columns, and creates 2 new indexs on 2 frequest accessed columns.

5 Conclusions

To reduce index maintenance, this paper has propose the asynchronous adaptive delay tolerant index cache using delta cell. This method has some features such as dynamic index management and fine-grained controls. This is a new method to improve the database performance.

Acknowledgement

This work was supported by the Doctoral Fund of University of Jinan (XBS1237), the Shandong Provincial Natural Science Foundation (ZR2014FQ029), the Shandong Provincial Key R&D Program (2015GGX106007), the Teaching Research Project of University of Jinan (J1344), the National Key Technology R&D Program (2012BAF12B07), and the Open Project Funding of Shandong Provincial Key Laboratory of Software Engineering (No. 2015SE03).

References

[1] Radoslaw Boronski and Grzegorz Bocewicz. Relational database index selection algorithm. In *Computer Networks*, pages 338–347. Springer, 2014.

[2] Harumi Kuno and Goetz Graefe. Deferred maintenance of indexes and of materialized views. In *Databases in Networked Information Systems*, pages 312–323. Springer, 2011.

[3] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[4] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, Chandrasekaran Mohan, Hamid Pirahesh, and Berthold Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 718–729. VLDB Endowment, 2003.

[5] Dan RK Ports, Austin T Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. Transactional consistency and automatic management in an application data cache. In *OSDI*, volume 10, pages 1–15, 2010.

[6] Priya Gupta, Nickolai Zeldovich, and Samuel Madden. A trigger-based middleware cache for orms. In *Middleware 2011*, pages 329–349. Springer, 2011.

[7] Shahram Ghandeharizadeh and Jason Yap. Cache augmented database management systems. In *Proceedings of the ACM SIGMOD Workshop on Databases and Social Networks*, pages 31–36. ACM, 2013.

[8] Shahram Ghandeharizadeh and Jason Yap. Gumball: a race condition prevention technique for cache augmented sql database management systems. In *Proceedings of the 2nd ACM SIGMOD Workshop on Databases and Social Networks*, pages 1–6. ACM, 2012.

[9] Ying Ming Gao, Jia Huo, Kai Zhang, and Xian Zou. Database index management, February 13 2012. US Patent App. 13/371,577.

[10] Peter A Carlin, Per-Ake Larson, and Jingren Zhou. Asynchronous database index maintenance, March 20 2012. US Patent 8,140,495.

[11] Meiyalagan Balasubramanian and Rohit Sabharwal. Dynamic integrated database index management, July 16 2013. US Patent 8,489,565.

[12] John Martin Whitehead, Subrahmanyeswar Vadali, and Kalur Sai Kishan. Database index monitoring system, January 7 2014. US Patent 8,626,729.

[13] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. Database tuning advisor for microsoft sql server 2005: demo. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 930–932. ACM, 2005.

[14] Gary Valentin, Michael Zuliani, Daniel C Zilio, Guy Lohman, and Alan Skelley. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 101–101. IEEE Computer Society, 2000.

[15] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.