Technical Brief

Transparent SQL Query Caching

A solution that requires no changes to existing applications or database servers

Introduction

Today's complex business applications typically contain massive volumes of data and serve large numbers of users—a trend that drives performance requirements that are increasingly difficult to attain. Caching technology is frequently called upon to improve performance in many contexts.

While most relational databases contain an in-memory cache, these implementations are largely ineffective since they are coarse grained, lack instrumentation, and use global locks. Also, they clear the table-based cache frequently, for every entry modification. These constraints have led application developers to use generic caching frameworks and adapt them for SQL queries, a process that requires significant development resources, both to implement invasive changes to the application and to support ongoing maintenance.

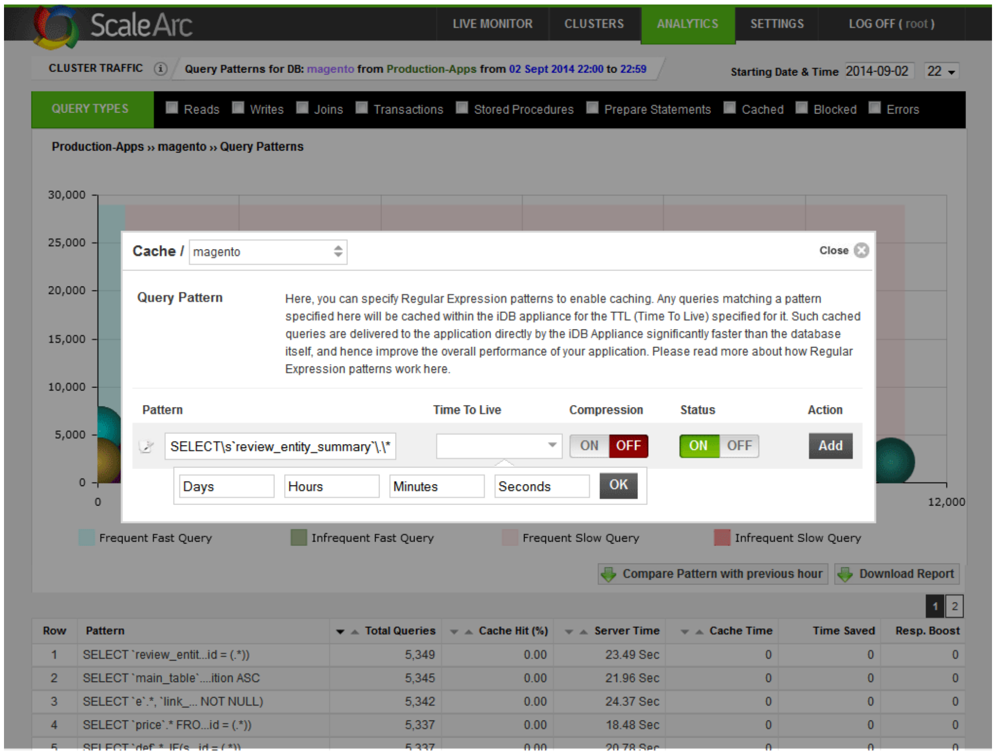
Caching technology is frequently called upon to improve performance in many contexts.

ScaleArc provides a transparent query result-set caching solution for databases that requires no changes to existing applications or database servers and does not require any development effort to realize all of the caching benefits. ScaleArc's SQL caching is an agentless approach that uses an in-memory key-value store (i.e., a NoSQL database) to store repetitive query responses, providing very fast response to subsequent matching queries. ScaleArc caching works at the query pattern level; these SQL query patterns are generated at wire speed from de-duplication of ScaleArc's centralized logging of all SQL queries flowing through ScaleArc. The cache can operate in various ways, including through the API.

Transparent SQL Query Caching

ScaleArc's patented SQL query cache is the world's first transparent NoSQL technology, which provides any database-driven application NoSQL-like performance without any application changes. ScaleArc's SQL query cache is fully ACID compliant and controllable by users as they see fit. By default, no caching is enabled, ensuring the application performs identical to how it would if it were working directly with the database server.

ScaleArc's cache provides users the ability to cache read-only SQL query responses in an extremely granular manner and to control how and when the cache expires using various methods. Users can choose among Time-to-Live (TTL) invalidation, API or query comment/hint-based invalidation, or automatic invalidation based on incoming writes or updates.



Query Patterns for DB: magento from Production-Apps from 02 Sept 2014 22:00 to 22:59

Starting Date & Time 2014-09-02 22

QUERY TYPES: Reads, Writes, Joins, Transactions, Stored Procedures, Prepare Statements, Cached, Blocked, Errors

Production-Apps ›› magento ›› Query Patterns

Cache / magento

Query Pattern: Here, you can specify Regular Expression patterns to enable caching. Any queries matching a pattern specified here will be cached within the iDB appliance for the TTL (Time To Live) specified for it. Such cached queries are delivered to the application directly by the iDB Appliance significantly faster than the database itself, and hence improve the overall performance of your application. Please read more about how Regular Expression patterns work here.

Pattern	Time To Live	Compression	Status	Action
SELECT\s`review_entity_summary`\.*		ON [✓ OFF]	[✓ ON] OFF	Add

Days, Hours, Minutes, Seconds, OK

🟦 Frequent Fast Query 🟩 Infrequent Fast Query 🟪 Frequent Slow Query 🟥 Infrequent Slow Query

Row	Pattern	Total Queries	Cache Hit (%)	Server Time	Cache Time	Time Saved	Resp. Boost
1	SELECT `review_entit...id = (.*))	5,349	0.00	23.49 Sec	0	0	0
2	SELECT `main_table`....ition ASC	5,345	0.00	21.96 Sec	0	0	0
3	SELECT `e`.*, `link_... NOT NULL)	5,342	0.00	24.37 Sec	0	0	0
4	SELECT `price`.* FRO...id = (.*))	5,337	0.00	18.48 Sec	0	0	0
5	SELECT `def`.* IF(s...id = (.*))	5,337	0.00	20.78 Sec	0	0	0

Figure 1. Convert your application analytics patterns into a Regex rule to match a specific query.

ScaleArc makes it very easy to identify which read-only SQL queries are good candidates for caching and create cache rules to see how effective the cache rules are in improving performance and reducing database load. ScaleArc's analytics displays provide administrators with a view of all the database queries being executed on the database cluster and lets them configure cache rules from the query patterns provided. ScaleArc shows this data in a simple, pattern-based view, which groups queries with the same SQL syntax but different variables together, and makes it easy to see which queries are resource intensive and frequent.

ScaleArc's cache rules are based on PCRE (Perl Compatible Regular Expressions), the most common Regular Expressions (Regex) format used today. But you don't need to know Regex to use ScaleArc. ScaleArc's analytics let you convert your application analytics patterns into a Regex rule to match a specific query.

For example, if you find a SELECT query, or a read-only stored procedure that queries relatively static data that is frequently accessed such as a Zip Codes table, you can click on that pattern in ScaleArc's analytics UI and choose to cache it for a user-defined period. This click creates a cache rule that matches all queries with the same SQL syntax but different variables. Once these queries are being served from cache, you will see a cache hit rate in the analytics view as well as the time saved by or performance improvement resulting from this cache rule.

Query Example

```
SELECT top 100 * FROM [PHONES] WHERE Phone>=209447 and Phone<=210447
```

Analytics Pattern Example

```
SELECT top (.*) * FROM [PHONES] WHERE Phone>=(.*) and Phone<=(.*)
```

Automatically generated Regular Expression Pattern Rule that will match the query with all variables

```
SELECT\stop\s(([+-.]|)[0-9.]+)\s\*\sFROM\s\[iDBTest\]\sWHERE\sPhone\>\=(([+-.]|)[0-
9.]+)\sand\sPhone\<\=(([+-.]|)[0-9.]+)
```

Figure 2. ScaleArc detects a SQL Query such as the one above, and a pattern is created from it, which can be automatically converted into a Regex rule with one click by the user.

Once a read query matches a cache rule, the result set of the query is stored in the native database protocol format in an in-memory NoSQL cache purpose-built by ScaleArc. The next time the application sends the same query, the stored response is served by ScaleArc, as long as the entry is still valid. Since the application sees the response in the native database protocol format, it thinks the response has come from the database rather than an intermediate NoSQL cache and processes the response as usual.

ScaleArc's cache can be expired / invalidated using three different methods, which suit a wide variety of use cases:

- Time-to-live (TTL)-based invalidation
- Automatic Cache Invalidation
- API-, Schedule-, or Query Comments-based Invalidation

Time-to-live (TTL)-based invalidation

TTL-based invalidation is the most commonly used cache expiration method, because it is functionally similar to how most NoSQL caches work, yet is transparent to the application. ScaleArc lets you pick the read queries you wish to cache and define how long the cache should be valid, from 1 second to 999 days. The flexibility of being able to pick a unique TTL for each query type or pattern make it possible to use the cache for a wide array of query types. Once a query has been cached using the TTL-based invalidation method, it will keep getting served entirely by ScaleArc for as long as the TTL is still valid. The queries cached using the TTL method can also be manually expired using the API or the query hints.

The following simple examples are use cases that can be handled using TTL-based invalidation:

- **Range / Sort / Multi-Variable / Multi-Row Queries**

  Such queries are typically significantly more computationally expensive than simple point select queries. They also fetch a lot more records at the same time and hence are extremely I/O intensive. A lot of web applications use such queries for fetching top records for a listing or home page, such as "Top Selling Items" on an eCommerce website or "Latest Headlines" on a news website. Since such data does not change frequently and is not transactional in nature but tends to be queried quite regularly, it's a good idea to cache such queries for between 1 to 10 minutes based on your business's "freshness" requirement. For example, a massive multiplayer gaming customer cached the query that fetches "Top 50 Players" out of a table of more than 20 million records with a TTL of 5 minutes. The customer achieved a 99% cache hit rate on that query and reduced their server I/O by more than 20% with one simple cache rule. The 5-minute time was chosen because that was the average amount of time users spent on a game session, and the "Top 50" list was shown on the screen after the game session ends.

- **Point Selects for Infrequently changing data / meta-data**

  Many applications use a lot of point queries to fetch details on items in a table that either never change or change very infrequently. A lot of metadata, which is used by applications to form correlations between different pieces of data, falls into this category. Some examples are:

  - queries that fetch the city name based on a provided zip code, could be cached for 24 hours or more and provide over 99.9% offload
  - queries that fetch the full name of a company based on a stock quote lookup, such as MSFT translating to Microsoft, Inc. can also be cached for 12 hours or more, as stock codes don't change within a day, and again achieve 99.9% or more hit rate
  - queries that fetch the details of a particular SKU from a products table in an eCommerce site could be cached for as little as 1-5 minutes for sites where item information changes frequently and still achieve 80% offload or greater in most cases –sites where the information changes less frequently could use a longer time and gain a larger offload

- **Large Aggregation / Reporting Queries on Old Data**

  A lot of applications run historical reports where they refer to data older than the current date, and that data doesn't change at all once it's been written. Typically, such query patterns are fairly resource intensive for the database to execute and lead to a lot of disk I/O as well. An example of this model could be queries that fetch historical graph data for stock quotes or a query that fetches details on all orders processed at a previous date. These kind of queries are used by managers, admins, and analysts frequently and are accessed by many people in the same day. By caching such queries for 24 hours or more, you could offload a massive amount of burst database I/O and speed up viewing of such information by the users.

ScaleArc's auto cache invalidation feature uses the transparent NoSQL technology by extracting metadata from the query and tagging the cache objects used to associate cache entries with invalidation queries.

Automatic Cache Invalidation

ScaleArc has introduced a new approach to cache invalidation to the industry – a method for automatically invalidating cache entries that enables true ACID-compliant caching. This method tracks data changes by extracting metadata from update or delete queries or from within SQL comments. ScaleArc's auto cache invalidation feature uses the transparent NoSQL technology by extracting metadata from the query and tagging the cache objects used to associate cache entries with invalidation queries. With this invalidation method, ScaleArc can guarantee that its cache will not serve stale data. This feature significantly increases the number of use cases where you can apply caching without the risk of data inconsistency challenges that can result from using TTL-based invalidation.

The ScaleArc analytics UI helps identify the queries that belong to the same table and access the same column. Users can then add invalidation patterns to an invalidation group that manipulates the same table and column. ScaleArc then has a cache invalidation group consisting of read queries or stored procedures as well invalidation queries or stored procedures all using a common column. ScaleArc supports string, boolean, long, double, short, byte, binary, decimal, byteArray, date, time, timestamp, and CharacterStreams as column types.

For example, consider a product catalog table with a column of “product_id.” The application issues select or update queries to retrieve or update data from the table using “product_id.” ScaleArc uses the “product_id” value as the metadata to tag the cache objects internally.

Cache Query

```
SELECT * FROM catalog.product_details WHERE catalog.product_id=1
```

Invalidation Query

```
UPDATE catalog SET name = 'ScaleArc‘ WHERE product_id = 1
```

Figure 3. An auto cache invalidation group can be formed by the above select and update queries since both of them use the same column “product_id.”.

Once the cache rules are created and grouped, ScaleArc will add cache objects created by the select calls with the metadata value extracted from the column location. When the application modifies data with the update or insert queries, ScaleArc will extract the metadata values from the column location. Using the column value or metadata, ScaleArc will invalidate only those cache items where the column value or metadata is an exact match. Subsequent select queries with the same invalidated column value(s) will get a cache miss and will be sent to the database. The invalidation mechanisms can also be triggered via API or as part of a SQL comment that includes the metadata for the column values.

The following use cases are supported using the auto cache invalidation-based method:

- **Shopping cart data**

  Shopping cart data is tracked for a user in an eCommerce application on every page load. Users browse through a lot of items before the final checkout, which puts significant load on the database. It is widely known that having slow load times impacts sales, and querying cart data from the database can slow web page performance. Up to 40% of web users abandon an eCommerce site if page load times exceed 3 seconds. Shopping cart data, despite being unique for each user, can now be cached with auto cache invalidation since ScaleArc ensures data consistency and reflects the new items in ScaleArc’s cache as soon as the cart is modified.

- **User profile data**

  User profile data, that is personal data associated with a specific user, is typically the most accessed data for the majority of applications. User profile databases need to have 100% uptime because they are validating users before any transactions can be performed. Users seldom modify their profiles, but when they do – such as update a password – the app must immediately reflect the changed data. The user profile database lends itself very well to auto cache invalidation since each user has a unique id within the database. Data tracking within auto cache invalidation keeps the user profile query cache up to date and significantly offloads the database from processing repetitive queries.

- **Auction data**

  eCommerce sites where users can buy and sell items in an auction require the auction price and associated data such as current bidders to be fetched and updated frequently. The page load performance and the accuracy of auction data are extremely important for these sites. Database resources are heavily taxed towards the end of the auction when many more users are watching an item and starting a bidding war. Each item has a unique identifier that is used to query the data, making it a prime candidate for auto cache invalidation.

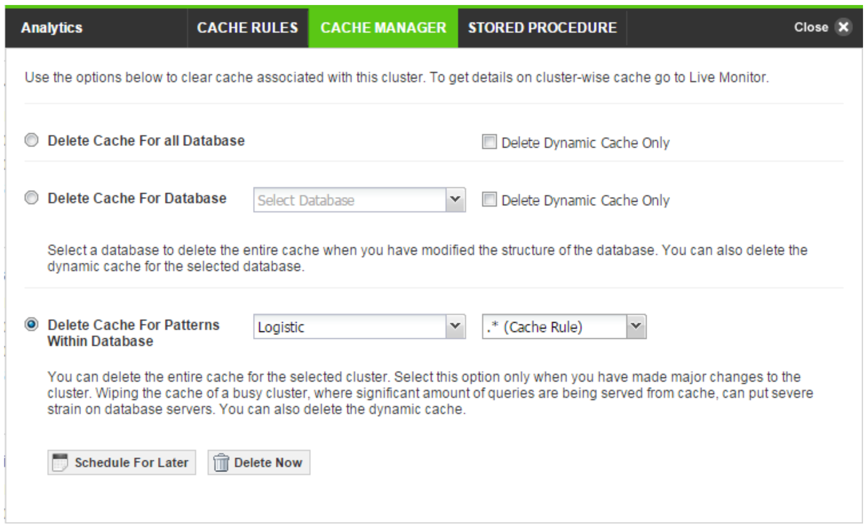
Auto cache invalidation can also be useful where point selects are prevalent but cannot be cached using the TTL-based method.

API-, Schedule-, or Query Comments-based Invalidation

This method of cache invalidation is used less often, but provides for significantly more customizability and complex caching use cases than the other two methods. Using ScaleArc's cache management APIs, or query hints, you can customize when cache is created, and you can clear or invalidate cache programmatically either for an individual query or for a whole group of queries.

ScaleArc's RESTful cache management API calls, or the cache management UI which is powered by the same APIs, let you clear cache at multiple levels. You can choose to clear or invalidate queries related to a specific cache pattern within a specific database, or you can clear the cache for the whole database or a whole cluster with a single API call. This flexibility makes it easy to refresh the cache at custom intervals or after a major data change operation. For example, you could immediately clear the cache for all queries for a Zip Code table if the postal service issues a new list.

Using ScaleArc's cache management APIs, or query hints, you can customize when cache is created, and you can clear or invalidate cache programmatically either for an individual query or for a whole group of queries.



Use the options below to clear cache associated with this cluster. To get details on cluster-wise cache go to Live Monitor.

○ Delete Cache For all Database ☐ Delete Dynamic Cache Only

○ Delete Cache For Database Select Database ☐ Delete Dynamic Cache Only

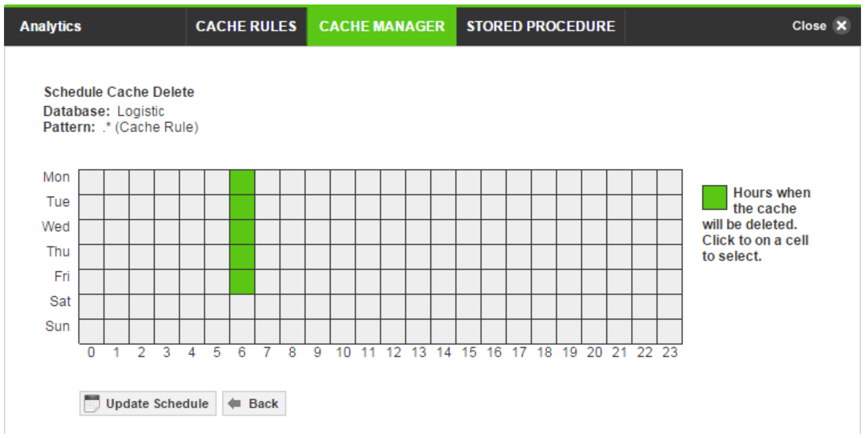
Select a database to delete the entire cache when you have modified the structure of the database. You can also delete the dynamic cache for the selected database.

● Delete Cache For Patterns Within Database Logistic .* (Cache Rule)

You can delete the entire cache for the selected cluster. Select this option only when you have made major changes to the cluster. Wiping the cache of a busy cluster, where significant amount of queries are being served from cache, can put severe strain on database servers. You can also delete the dynamic cache.

Figure 4. ScaleArc's RESTful cache management API calls, or the cache management UI which is powered by the same APIs, let you clear cache at multiple levels.

ScaleArc also provides a cache-management scheduler, which lets you clear cache at pre-defined intervals. This is very useful to automate clearing cache when data refresh happens at predictable windows. For example, if a logistics platform receives new location data every morning at 6 AM on workdays, you could schedule the system to automatically clear all cache related to the logistics database, or to a specific table, automatically every day at 6 AM using the ScaleArc cache management scheduler.



Schedule Cache Delete
Database: Logistic
Pattern: .* (Cache Rule)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Mon							🟩																	
Tue							🟩																	
Wed							🟩																	
Thu							🟩																	
Fri							🟩																	
Sat																								
Sun																								

🟩 Hours when the cache will be deleted. Click to on a cell to select.

Figure 5. Clear cache at pre-defined intervals with ScaleArc's cache-management scheduler.

Last, but not the least, ScaleArc lets you control cache functions using query hints or comments. This method is very useful for immediately caching or invalidating specific queries from within the application. This approach is useful when developers want to control ScaleArc's cache just as granularly as they would a traditional NoSQL cache but do so with minimal changes to the application – and no need to deal with additional NoSQL APIs.

ScaleArc extends the existing SQL comments functionality that exists within all databases and uses that to control caching functions. This architecture is extremely beneficial from a compatibility standpoint, because even if the application ever has to connect directly to the database server without ScaleArc, the application will still work as expected without any errors – it just won't have the benefit of the ScaleArc cache.

/*ttl(60)*/SELECT * from CountriesList - This comment will cache the specified query for 60 seconds.

/*wipe*/SELECT * from CountriesList - This comment will wipe or invalidate the query specified and get the current results from the database servers.

/*nocache*/SELECT * from CountriesList - This comment will skip the cache for the query specified and get the current results from the database servers.

Figure 6. By simply prepending a SQL query with the above comments, you can cache, invalidate, or skip cache for certain queries.

ScaleArc has been granted a patent on "Method and system for transparent database query caching". **ScaleArc Patent #8,543,554**

Advantages over Memcached

To accelerate application performance, app developers are using generic caching frameworks and adapting them for SQL queries, a process that requires significant development resources, both to implement invasive changes to the application and to support ongoing maintenance.

NoSQL-based caching technologies have proliferated as a result. For example, Memcached is a popular generic caching framework that is sometimes used for SQL query caching. While Memcached can address some performance and scaling issues and can cache objects other than queries, it has several major flaws when used in the context of SQL query caching:

- Memcached requires significant changes to each application, limiting its deployment to organizations that are willing to commit substantial engineering resources.
- Application developers need to manually manage the cache, adding complexity.
- Memcached is not designed specifically for SQL, making some tasks cumbersome, such as deletion of large swathes of cache.
- Memcached does not provide cache persistence across server reboots.
- Memcached does not provide cache usage statistics on a per-query-pattern basis.
- Large installations of Memcached suffer from a high TCP connection count, leading to scaling difficulties.
- Memcached does not continuously replicate cache to a failover server.
- Memcached does not authenticate users, possibly reducing the security posture of the application stack.

Summary

Caching has long been a popular technique in computing to shorten data access times, to reduce latency and to improve input/output operations. In the context of applications, caching improves application performance. However, as we previously explored, caching approaches in databases have limited advantages: invasive changes to the application are required and a fundamental need to support ongoing maintenance. Developers have turned to Memcached, a free popular open source distributed memory object caching system, for database caching, but it too has several major flaws.

ScaleArc offers a simple way to leverage the performance advantages of caching that requires no changes to existing applications or database servers and does not require any development effort. For more information on ScaleArc's caching technology, please visit **http://www.scalearc.com/how-it-works/performance-features/transparent-in-memory-query-caching**.

ScaleArc

2901 Tasman Drive, Suite 205
Santa Clara, CA 95054
Phone: 1-408-780-2040
Fax: 1-408-427-3748
www.scalearc.com

02/23/15