Implementation of a Query Plan Cache in AsterixDB

Sushrut Borkar

July 2023

Abstract

When a query is submitted to a database system, it typically goes through a sequence of compilation steps before it can be run. If the same query is submitted repeatedly, it must be recompiled each time. A query plan cache can reduce this time by skipping compilation steps for previously seen queries. We describe the implementation of a query cache in AsterixDB, a big data management system with a NoSQL style data model. Finally, we test performance improvements due to caching.

1 Introduction

Before a database system can execute a query, it must transform the query into an efficient query execution plan by passing it through a parser, rewriter, and optimizer. These steps have significant overhead, and if the same query is subsequently submitted, it must pass through these steps again. To avoid this cost, these execution plans can be stored in a query plan cache. If the same or a similar query is reissued, compilation can be skipped and the cached plan can be used to satisfy the query [5]. Many commercial DBMSs implement such caches [6, 7, 8].

This thesis describes the implementation of such a cache in AsterixDB. Section 2 gives an overview of the architecture and compilation flow of AsterixDB. Section 3 describes the implementation details of the cache. Section 4 details the syntax for fine-tuning the behavior of the query cache. Section 5 evaluates performance improvements due to caching. Section 6 concludes the paper and describes possible future work.

2 Overview of AsterixDB

AsterixDB is a big data management system (BDMS) designed to store, manage, query, and analyze large amounts of unstructured data [1]. It features a flexible data model fitting a variety of use cases, a query language for JSON data (SQL++), and LSM-based storage and indexing. It is also capable of scaling outward on shared-nothing commodity clusters. Datasets in AsterixDB are stored as B+ trees indexed by the primary key of the dataset, which is also used to hash-partition the dataset across cluster nodes. A single cluster controller receives user requests and coordinates work among the nodes.

When a query string is received by AsterixDB, it is first passed through a query parser, which lexically analyzes it and converts it into an abstract syntax tree (AST). This is then passed to the rewriter to create a rewritten AST. Next, the rewritten AST is passed to a translator that converts it into a logical plan. This plan is composed of logical operators, and is used for type inference and checking. The logical plan is then passed to the optimizer, which rewrites the plan into one that can execute more efficiently. While previous versions of AsterixDB used a rule-based optimizer, it has recently added cost-based optimization (CBO), which maintains statistics about datasets to estimate costs of different logical plans and select the most efficient one. Finally, the optimized plan is passed to a job generator, which turns it into a Hyracks job specification that can be run by the Hyracks execution engine. More details on compilation in AsterixDB can be found in [2].

3 Implementation

To support plan caching in AsterixDB, we alter existing query compilation flow. After the query is parsed and converted to an AST, we construct a string version of the AST and look up this string in the query cache. If the query is not found in the cache, we proceed with usual compilation, and then store the generated Hyracks job specification in the cache for future use. On the other hand, if the query is found, we can skip rewriting, translation, optimization, and job generation, and directly run the cached job specification.

3.1 Cache Design

To support efficient lookups, the cache is implemented as a concurrent hash map with keys and values. While the most important part of the cache key is the string version of the AST, we must include additional information in the key to avoid incorrect lookups. In fact, any parameter that can affect the job specification that a given query would produce must be made part of the key. In total, five attributes are included:

1. The *AST string*, as described above. Using this for lookups is advantageous compared to using the raw query string, because the AST is resilient to minor changes in the query, such as spaces or empty lines, resulting in more cache hits.

2. The *session configuration*, containing information such as the client type, preferred output format, and execution flags. If these are changed, a given query could produce a different job specification.

3. The collection of used *SET statements*, which can override certain configuration parameters, such as operator memory budgets, whether to use index-only plans, or whether to use CBO.

4. The *active dataverse*. Dataverses are the top-level organizing concept in AsterixDB, within which datasets, functions, and types can be created.

5. The *result set ID*. If a single request submitted to AsterixDB includes multiple queries, each is given a different result set ID to distinguish among them.

Similarly, we include three attributes in the cache value:

1. The *Hyracks job specification* to be submitted to the Hyracks execution engine.

2. *Cached warnings*. AsterixDB may issue warnings during compilation steps, which we skip for cache hits. As a result, queries answered using the cache fail to issue compilation-related warnings. To get around this, we cache warnings issued during compilation, and reissue these warnings for cache hits.

3. A *lock*. AsterixDB does not allow running a given job specification from multiple threads at the same time. As a result, if two threads look up the same cache entry and run its

job specification, the system fails. To prevent this, we include a lock in each cache value which a thread must acquire before using its job specification and release after the job is finished running. If the lock is held by another thread, we recompile the query into a new job specification instead of blocking.

Additionally, to prevent queries that raise runtime errors from being cached, we wait until successful completion of a job before inserting into the cache. The cache architecture is illustrated in Figure 1.

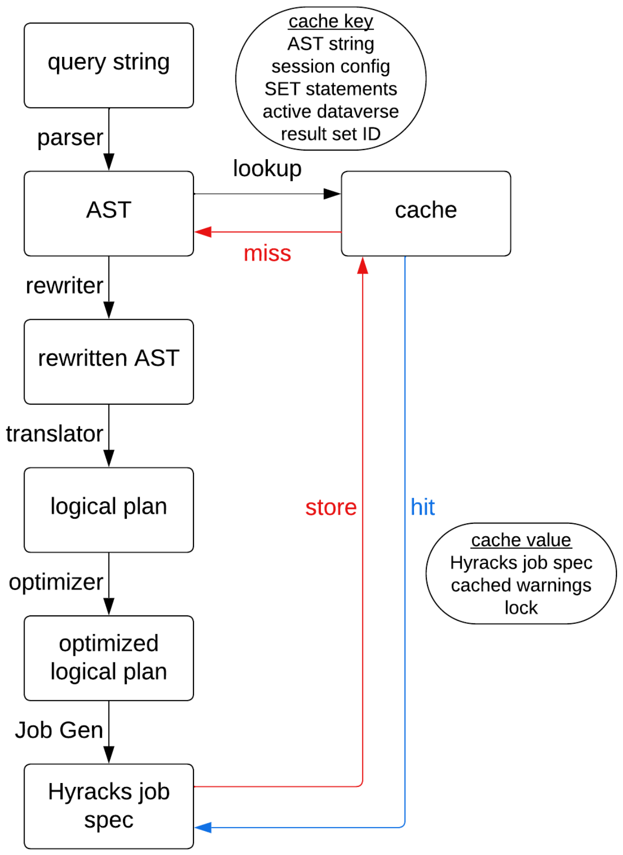


Figure 1: Query cache architecture diagram.

3.2 Cache Invalidation

Whenever some event changes the job specification that some query should produce, affected cache entries should be invalidated. For example, if a new index is created that might speed up execution of a cached query, the query should be removed from the cache. Consequently, if the query is resubmitted, it can be recompiled and the index can be used in the resulting job specification.

Currently, we invalidate all cache entries on such events. A list of commands that invalidate the cache can be found in Table 1. A future improvement is to keep track of the dependencies of each cached query so that only affected cache entries are invalidated. This could be done efficiently by maintaining a kind of "secondary index" mapping from dependencies of a given query (e.g.

CREATE_INDEX	VIEW_DROP	ANALYZE
INDEX_DROP	FUNCTION_DROP	ANALYZE_DROP
DATAVERSE_DROP	LIBRARY_DROP	FULL_TEXT_FILTER_DROP
DATASET_DROP	SYNONYM_DROP	FULL_TEXT_CONFIG_DROP
NODEGROUP_DROP	DROP_FEED	DISCONNECT_FEED
	REBALANCE	

Table 1: Commands that invalidate the query cache.

datasets, indexes, functions) to query cache keys, so that when a dependency is affected, relevant cache entries can be found easily.

4 User Experience

The query plan cache is enabled by default in AsterixDB and clients do not have to do any additional work to use it. However, we provide some commands to help users control and tune cache behavior as desired. We introduce two new `SET` statements:

1. `` SET `compiler.querycache.bypass` "true"; ``
2. `` SET `compiler.querycache.clear` "true"; ``

The first statement above forces a query to ignore the cache, avoiding cache lookups and insertions. The second statement clears the entire cache. Both commands are useful to users that want some queries to be recompiled even when cached job specifications are available. Since AsterixDB can also be accessed from an HTTP API, we have added an additional API parameter `bypass_cache` that serves the same purpose as the first `SET` statement above but is more convenient for some users.

To prevent the cache from growing indefinitely, we limit the cache size to 1024 entries by default. If the cache is full, the LRU (least recently used) replacement algorithm is used to free space for new cache entries. To accommodate users who would like the cache capacity to be larger (to get more cache hits) or smaller (to use less memory), we have added a new parameter `query.cache.capacity` that the user can set by editing the `cc.conf` file, which stores information used by AsterixDB when starting up.

5 Evaluation

We evaluate performance improvements from the query cache with two experiments. In the first experiment, we run queries from CH2 [3], a benchmark based on TPC-C and TPC-H designed to evaluate hybrid transactional/analytical document databases. In the second experiment, we test caching improvements as the number of joins and number of indexes are changed.

5.1 Experimental Setup

We selected 17 queries from the CH2 benchmark and ran them with and without the query cache. We generated three versions of the CH2 data having different sizes to test improvements due to caching as data size is increased. Versions with 1 CH2 warehouse, 4 warehouses, and 16 warehouses were generated, resulting in total data sizes of 160 MB, 600 MB, and 2.3 GB respectively. Datasets were indexed by their primary key fields, and appropriate secondary indexes and multi-valued indexes [4] were also created. For the second experiment, we generated 6 datasets each having 50,000 records (8 MB) to test performance as the number of joins in a query is increased, and 1 dataset having 2,000,000 records (350 MB) to test performance of a simple select query as the number of indexed fields is increased. All experiments were performed locally on machine with an Intel i7-12700H, 40 GB of RAM, and a 512 GB SSD. The full set of queries and experiments can be found at `https://github.com/sushrutborkar/cache-benchmark`.

```
SELECT    ol.ol_number,
          SUM(ol.ol_quantity) AS sum_qty,
          SUM(ol.ol_amount) AS sum_amount,
          AVG(ol.ol_quantity) AS avg_qty,
          AVG(ol.ol_amount) AS avg_amount,
          COUNT(*) AS count_order
FROM      orders o, o.o_orderline ol
WHERE     ol.ol_delivery_d > "2014-07-01 00:00:00"
GROUP BY  ol.ol_number
ORDER BY  ol.ol_number;
```

(a)

```
SELECT *
FROM   R1, R2, R3, R4, ...
WHERE  R1.fk = R2.id
  AND  R2.fk = R3.id
  AND  R3.fk = R4.id
  AND  ...;
```

(b)

```
SELECT R.id
FROM   R
WHERE  R.f1 /*+ use-index (ind1) */ = $a
  AND  R.f2 /*+ use-index (ind2) */ = $b
  AND  R.f3 /*+ use-index (ind3) */ = $c
  AND  R.f4 /*+ use-index (ind4) */ = $d;
```

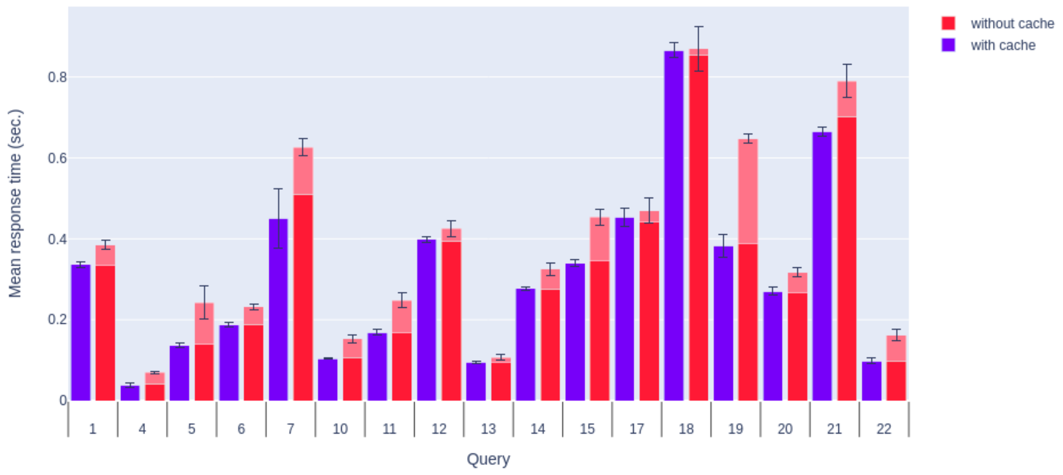
(c)

Figure 2: Three of the SQL++ queries used for performance testing. Query (a) is the first query from the CH2 benchmark. Query (b) is used to test performance as the number of joins is increased. Query (c) is a simple select query that we use to test performance improvements as indexes are built on fields `f1` to `f4`. The values of `$a`, `$b`, `$c`, and `$d` are chosen uniformly at random from 1 to 10, and the `use-index` flag is added to force usage of available indexes.

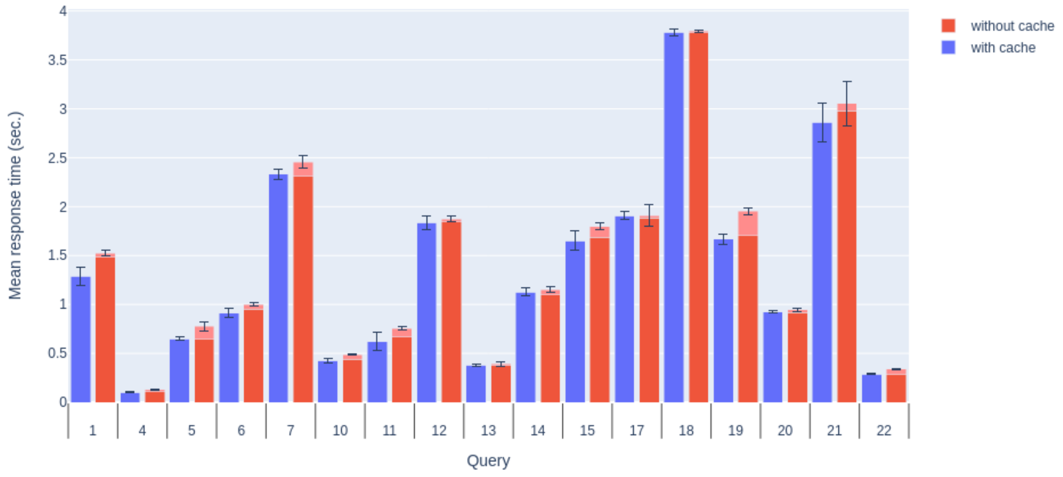
5.2 Results

The CH2 queries were executed for 12 runs, while interleaving with-cache and without-cache runs. Measurements from the first 2 runs were discarded as they were used to warm up the database and to insert all queries into the cache, resulting in 5 runs with the cache and 5 without. From these, we calculated the mean response time (as measured by clients) and compilation time (as measured by the database) for each query.

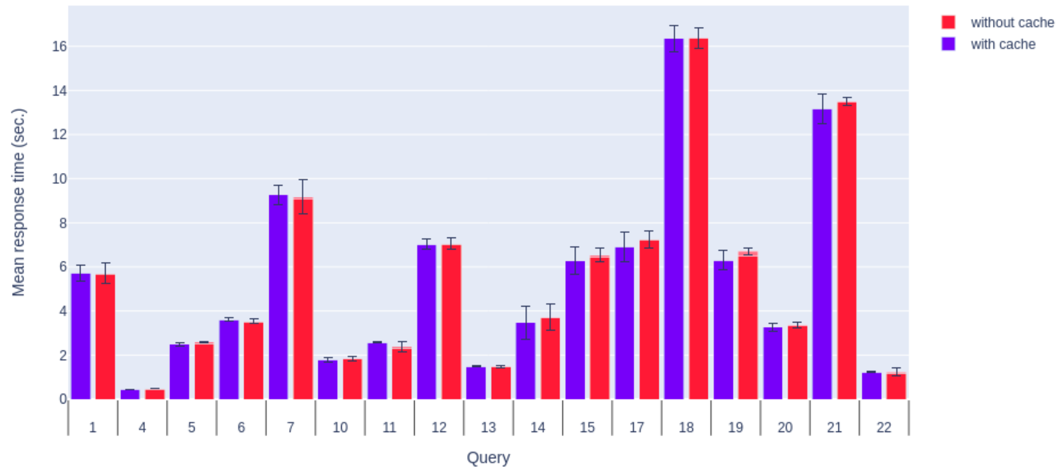
The measurements for each data size are shown in Figure 3. As can be seen, caching provides a significant speedup for small data sizes (1 warehouse), but the improvement diminishes as data size increases, since execution time takes up a much bigger part of total response time than compilation



(a) Results with 1 CH2 warehouse.

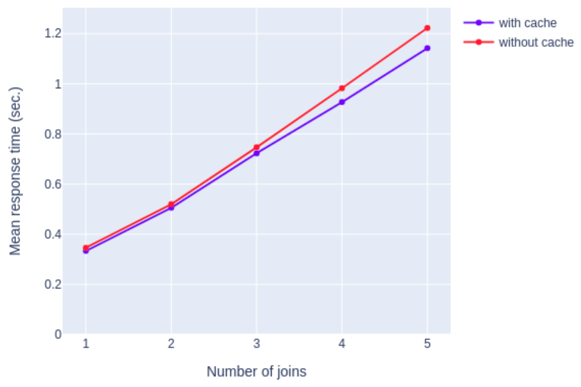


(b) Results with 4 CH2 warehouses.

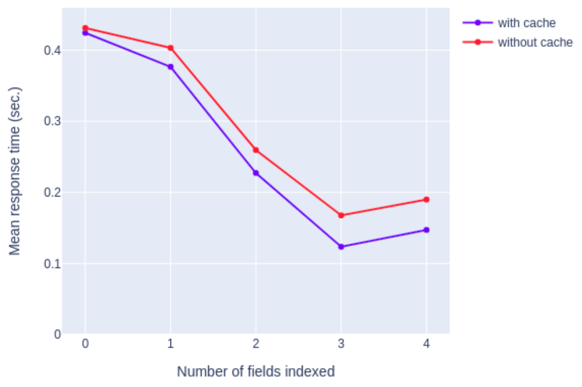


(c) Results with 16 CH2 warehouses.

Figure 3: CH2 benchmark results. Compilation time is shown in light colors.



(a) Join test.



(b) Index test.

Figure 4: Results of the second experiment.

time. The query cache also brings compilation time for queries answered from the cache down to hundreds of microseconds, and as a result this time can barely be seen in the plot. This is not surprising, considering that every compilation step besides parsing is skipped for such queries. The cache is most useful for queries where compilation time takes up a very large proportion of overall response time, such as queries 5, 11, and 19. On average, the cache improved response times by 22.5% for 1 warehouse, 8.9% for 4 warehouses, and 1.7% for 16 warehouses.

Results of the second experiment, where we tested the benefit of the cache as the number of joins and number of indexes changes, are shown in Figure 4. To estimate selectivities, CBO issues small sample queries during compilation. When the number of joins or number of indexes is increased, more of these small queries are issued, increasing compilation time. Therefore we would expect caching to provide greater benefit as the number of joins or indexes is increased, since more compile-time queries can be avoided. This is confirmed in Figure 4, as the time saved by caching can be seen to increase with the number of joins and indexes.

6 Conclusion

We described the implementation of a query plan cache in AsterixDB, as well as statements users can use to tune cache behavior. Based on the results of performance testing, the cache significantly improves response times, especially for queries running on small datasets where a large proportion of response time is spent on compilation. While this work focused on AsterixDB, the lessons learned can be applied to many other data systems.

Potential future work includes (a) a more fine-grained cache invalidation policy, (b) commands for easily visualizing current contents of the cache, (c) enabling running a single job specification concurrently from multiple threads so that no cache locking is necessary, and (d) memory consumption experiments to determine a better default cache capacity.

Acknowledgements

I would like to express my gratitude to Professor Michael Carey and Glenn Galvizo for their wise guidance and support throughout the past year. Also thanks to Vijay Sarathy for providing the CH2 queries.

References

[1] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, and T. Westmann. AsterixDB: A scalable, open source BDMS. *Proc. VLDB Endow.*, 7(14):1905–1916, Oct 2014.

[2] V. Borkar, Y. Bu, E. P. Carman, N. Onose, T. Westmann, P. Pirzadeh, M. J. Carey, and V. J. Tsotras. Algebricks: A data model-agnostic compiler backend for big data languages. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, page 422–433, New York, NY, USA, 2015. Association for Computing Machinery.

[3] M. Carey, D. Lychagin, M. Muralikrishna, V. Sarathy, and T. Westmann. CH2: A hybrid operational/analytical processing benchmark for NoSQL. In *Performance Evaluation and Benchmarking: 13th TPC Technology Conference, TPCTC 2021, Copenhagen, Denmark, August 20, 2021, Revised Selected Papers*, page 62–80, Berlin, Heidelberg, 2021. Springer-Verlag.

[4] G. Galvizo and M. J. Carey. Multi-valued indexing in Apache AsterixDB (SI DOLAP 2022). *Information Systems*, 113:102144, 2023.

[5] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a database system. *Found. Trends Databases*, 1(2):187–188, Feb 2007.

[6] IBM. Plan cache. `https://www.ibm.com/docs/en/i/7.4?topic=overview-plan-cache`.

[7] Microsoft. Query plan caching (entity SQL). `https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/query-plan-caching-entity-sql`.

[8] MongoDB. Query plan cache methods. `https://www.mongodb.com/docs/manual/reference/method/js-plan-cache/`.