In-Browser Interactive SQL Analytics with Afterburner

Kareem El Gebaly and Jimmy Lin
David R. Cheriton School of Computer Science
University of Waterloo, Ontario, Canada
{kareem.elgebaly,jimmylin}@uwaterloo.ca

ABSTRACT

This demonstration explores the novel and unconventional idea of implementing an analytical RDBMS in pure JavaScript so that it runs completely inside a browser with no external dependencies. Our prototype, called Afterburner, generates compiled query plans that exploit two JavaScript features: typed arrays and asm.js. On the TPC-H benchmark, we show that Afterburner achieves comparable performance to MonetDB running natively on the same machine. This is an interesting finding in that it shows how far JavaScript has come as an efficient execution platform. Beyond a mere technical curiosity, we demonstrate how our techniques can support interactive data exploration by automatically generating materialized views from a backend that is then shipped to the browser to facilitate subsequent interactions seamlessly and efficiently.

1. INTRODUCTION

Browser-based notebooks (i.e., Jupyter) have gained tremendous popularity with data scientists in recent years for a variety of reasons: The tight integration of code and execution output elevates the analytical process and its products to first class citizens, since the notebook itself can be serialized, reloaded, and shared. A recent development is the seamless integration of browser-based notebooks with scalable data analytics platforms, such that code written in a notebook cell can be executed on a potentially large cluster and the results can be further manipulated in the notebook. This integration, exemplified by the commercial Databricks platform that provides a notebook frontend to Spark clusters [11], allows data scientists to analyze large amounts of data in a convenient and flexible manner.

The advent of notebooks means that the browser has, in essence, become the "shell". However, in all implementations that we are aware of, the browser is simply a dumb rendering endpoint: all query execution is handled by backend servers. However, modern browsers are capable of so much more: they embed powerful JavaScript engines capable of running real-time collaborative tools, rendering impressive 3D scenes, and even running first-person shooters.

We asked: Is it possible to exploit modern JavaScript engines to build high-performance data management systems that run *completely in the browser*? And if so, what new opportunities does such a platform create? We tackle these two questions in turn: First, we present Afterburner, a prototype analytical RDBMS implemented in JavaScript that executes completely inside a web browser, taking advantage of in-memory columnar storage using typed arrays and query compilation into asm.js. Experiments with the standard TPC-H benchmark show that Afterburner achieves comparable performance to the analytical database MonetDB *running natively on the same machine*. Second, we describe a novel technique to support interactive data exploration. We address the common scenario in which a data scientist works with a "query template" and issues a sequence of queries that vary only in the predicates on certain fields. With a simple hint, Afterburner is able to automatically split such queries into two parts: a backend query that generates a materialized view and a frontend query against this view. Our implementation is novel in that the materialized view is shipped to the browser, so that all subsequent interactions occur locally. Experiments show that our approach yields a better user experience and brings additional benefits as well.

2. AFTERBURNER OVERVIEW

Afterburner is implemented as a JavaScript library, primarily designed to run inside a standards-compliant web browser. Data is loaded from flat files on the client's file system or via WebSockets as JSON objects from a remote server. All data are immutable and organized in a columnar layout in memory. Afterburner generates compiled query plans that exploit two JavaScript features: typed arrays and asm.js. We present key features of our system below:

In-memory columnar storage using typed arrays. Array objects in JavaScript can store elements of any type and are not arrays in a traditional sense (compared to C) since consecutive elements may not be contiguous and the array can dynamically grow or shrink. This flexibility limits the optimizations that the JavaScript engine can perform both during compilation and at runtime. In the evolution of JavaScript, it became clear that the language needed more efficient methods to quickly manipulate binary data: typed arrays are the answer. Typed arrays are comprised of buffers, which hold untyped binary data, and views, which impose a read context on the buffer. Typed arrays allow the developer to create multiple views over the same buffer, which pro-

SIGMOD'17, May 14 - 19, 2017, Chicago, IL, USA

ISBN 978-1-4503-4197-4/17/05…$15.00

DOI: http://dx.doi.org/10.1145/3035918.3058736

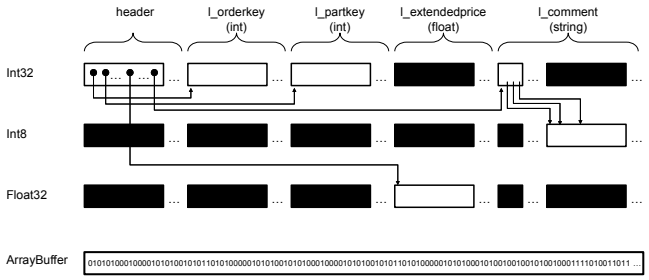


Figure 1: Illustration of the physical in-memory representation of the `lineitem` table from TPC-H. Different "views" (top three rows) provide access into the underlying JavaScript ArrayBuffer. Blackened boxes represent invalid data for that particular view.

vides a mechanism to interact with, for example, arbitrary C `structs`. Afterburner takes advantage of this feature to pack relational data into a columnar layout.

In Afterburner, each column is laid out end-to-end in the underlying buffer (which can be traversed with a view of the corresponding type). The table itself is a group of pointers to the offsets of the beginning of the data in each column. Figure 1 shows the physical memory layout storing the `lineitem` table from the TPC-H benchmark. A `lineitem` pointer serves as the entry point into a group of 32-bit integer pointers, which represent the offsets of the data in each column (`l_orderkey`, `l_partkey`, etc.). Our present implementation supports integers, floats, dates, and strings. For the first three types, values are stored as literals (essentially, an array). For a column of strings, we store null-terminated strings prefixed with a header of pointers to the beginning of each string, essentially a `(char **)` in C.

Query compilation into asm.js. In conjunction with typed arrays, Afterburner takes advantage of asm.js, a strictly-typed subset of JavaScript designed to be easily optimizable by an execution engine. At a high-level, Afterburner translates SQL into the string representation of an asm.js module (i.e., generates a physical query plan), calls `eval` on the code, which triggers AOT (Ahead-Of-Time) compilation and links the module to the calling JavaScript code, and finally executes the module (i.e., executes the query plan). The typed array storing all the tables (i.e., the entire database) is passed into the module as a parameter, and the query results are returned by the module.

The compiled query approach of Afterburner takes after systems like HIQUE [9], LegoBase [7], Proteus [6], and HyPer [10], which have recently popularized code generation for relational query processing. With the exception of targeting JavaScript (which to our knowledge is novel), our query compilation techniques are fairly standard. We use a template-matching approach to generating compiled queries; for example, simple SELECT-WHERE queries are converted into `for` loops over the appropriate ranges of the typed array holding the data. In our current prototype, there is no query optimization to speak of, as we only have a single hash-based physical plan for both joins and group-bys.

One well-known drawback of query compilation is that compiling generated code using a tool like `gcc` can overshadow its benefits for short-running queries. Much research has gone into alleviating this issue, such as the use of an intermediate representation like LLVM [10]. In the context

```
SELECT
 l_returnflag,
 l_linestatus,
 sum(l_quantity) as sum_qty,
 sum(l_extendedprice) as sum_base_price,
 sum(l_extendedprice*(1-l_discount)) as sum_disc_price,
 sum(l_extendedprice*(1-l_discount)*(1+l_tax)) as sum_charge,
 avg(l_quantity) as avg_qty,
 avg(l_extendedprice) as avg_price,
 avg(l_discount) as avg_disc,
 count(*) as count_order
FROM lineitem
WHERE l_shipdate <= date '1998-12-01' - interval '90' day
FREE  l_shipdate
GROUP BY l_returnflag, l_linestatus
ORDER BY l_returnflag, l_linestatus;
```

Figure 2: TPC-H Q1 showing the FREE clause extension that triggers Afterburner to generate materialized views to support interactive data exploration.

of our work, however, we have found compilation overhead to be negligible, primarily because compilation speed is already something that JavaScript engines optimize for, since all JavaScript code on the web is stored as text.

Instead of string-based SQL queries, Afterburner executes queries written using an API that is heavily driven by method chaining, often referred to as a *fluent* API. There is a very straightforward mapping from the method calls to clauses in a standard SQL query, so we can view the fluent API as little more than syntactic sugar. However, this query API is quite similar to DataFrames [1], an interface for data manipulation that many data scientists are familiar with today.

Automatic view materialization to support interactive data exploration. The work of data scientists often involves tight interaction cycles with the data, particularly for exploratory tasks. Viewed in terms of SQL, a common scenario is the execution of a sequence of queries that differ only in the predicates in the WHERE clause. Using Q1 from the TPC-H benchmark as an example (see Figure 2), the data scientist might be interested in the results of the same "query template", but with different date ranges (i.e., different predicates on the `l_shipdate` column). In fact, this is essentially what happens in dashboards and report generators that support interactive data exploration.

To support this scenario, we introduce an SQL extension in the form of a FREE clause (marked in red), which provides a hint to the SQL engine indicating that the data scientist is interested in follow-up queries that only differ in the predicates on the columns denoted in the clause (i.e., `l_shipdate` in this case). Afterburner recognizes this directive and optimizes such queries by automatically rewriting them into two component queries: (1) a *backend query* that generates a materialized view supporting efficient execution of all possible forms of the original query that differ only in predicates on the "freed" columns, and (2) a *frontend query* that contains the original query rewritten against this materialized view. In other words, query execution is split between the client and the server [3]. The materialized view generated by the backend query is automatically shipped over to the browser and stored as typed arrays in the in-memory columnar format discussed above. All subsequent queries of the same form are posed against this local materialized view. Note that our FREE clause goes beyond bound parameters in prepared statements (e.g., `l_shipdate = :date`) in that we support arbitrary predicates.

Our intuition is that while the backend query to generate

latency (ms)	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Q09	Q10	Q11
Afterburner	115	68	108	161	151	36	322	101	314	233	35
MonetDB	1269	36	242	121	141	431	161	113	235	126	42
latency (ms)	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
Afterburner	82	830	44	50	149	718	446	1951	62	284	95
MonetDB	119	243	40	74	316	185	171	209	86	369	118

Table 1: Query latency (ms) over five trials for the TPC-H queries, comparing Afterburner with MonetDB.

the materialized view might be more expensive, Afterburner is able to support much faster queries involving different predicates on the "freed" columns (by rewriting them into different frontend queries against the materialized view). As we describe later, the "freed" columns can be tied to interface widgets (e.g., slider bars) that support interactive data exploration. In our current implementation, columns that are part of a join condition cannot be freed; also disallowed are columns that are used in a complex condition in the WHERE clause or those involved in sub-queries.

Taking advantage of materialized views to optimize complex queries is of course a well-studied problem dating back decades [5, 4]. There are, however, substantial differences with our approach: in most previous work, materialized views are long-lived and carefully-considered optimizations by a database administrator, not built willy-nilly on an ad hoc and per-query basis—which is the approach that we take. For us, materialized views are transient and lightweight, precisely because they are shipped over to the browser and can be discarded when done. This approach exploits our in-browser JavaScript engine, which means that not only is the integration seamless, but subsequent interactions can happen without the backend. Our idea of "freeing" columns is related to the work of Koudas et al. [8] in "relaxing" join and selection queries, but their goal is to "back off" from queries that return empty results. Also related is the semantic prefetching idea of Bowman and Salem [2], who try to predict future queries based on past history; in our case, we require explicit hints from the user.

3. EVALUATION

We aim to answer two questions: First, what is the performance of Afterburner as a standalone analytical RDBMS? Second, how well does the automatic view materialization feature support interactive data exploration?

In-browser TPC-H benchmark. To answer the first question, we ran the standard TPC-H benchmark at a scale factor of 1 GB. This represents a rough upper bound on the amount of data that a commodity desktop or laptop can comfortably hold, in our case, a desktop with a 2.7 GHz Intel i5-5250U processor (4 cores, 3 MB of cache) and 8 GB of RAM, running Ubuntu 16. We compared Afterburner running in Mozilla Firefox (v50) with MonetDB (v11.23.13), an open-source analytical RDBMS. For MonetDB, we used the TPC-H test harness written by the developers of the system. For a fair comparison, MonetDB was configured to use a single thread, since code running inside a browser tab is single threaded. All our measurements were on a warm cache—we first ran each query five times, and then took measurements over the next five trials. For Afterburner, the measured latency includes query compilation overhead and all data are explicitly loaded in memory. For MonetDB, all data are cached in the underlying OS buffer caches.

Table 1 shows the latency of all 22 TPC-H benchmark queries comparing Afterburner and MonetDB. MonetDB finishes all queries in 4.8s, while Afterburner finishes all queries in 6.4s (only 33% slower). MonetDB is faster than Afterburner in 11 out of the 22 queries. These results show the feasibility of running an analytical RDBMS completely inside the browser. We find Afterburner's performance quite impressive, given the constraints of both the language (JavaScript) and the execution environment (a browser).

Interactive data exploration with TPC-H queries. To answer the second question, we considered a data warehousing scenario defined by TPC-H data at scale factor 100 GB. In our evaluation, the backend is MonetDB running on a server with dual 8-core Intel Xeon E5-2670 processors (2.6 GHz) with 256 GB of memory on Ubuntu 14.04, configured to take advantage of all available hardware resources. The frontend client is the same desktop described above.

For evaluation, we considered a subset of TPC-H queries. For each, we generated variant queries associated with the columns to "free", in the sense described above. This is shown in Table 2. For example, for Q5 we can free two columns: `r_name`, `o_orderdate`, yielding Q5a and Q5b. In principle, Afterburner can free more than one column at a time, which our implementation supports. In practice, however, there may not be sufficient memory at the client to store the materialized views for multiple columns.

For each variant query, Afterburner is able to construct corresponding backend queries and frontend queries as described previously. The execution time of these frontend (FE) and backend (BE) queries are shown in Table 2 (3rd and 6th columns). For reference, the execution time of the original query is also shown, in the 6th column under the original query rows. Finally, we show the size of the materialized view and the time taken to copy the data into the browser. All measurements were on a warm cache and we report averages over five trials.

Going back to our running example of Q1 from Figure 2: from the table we see that the unmodified query takes 34s, while the backend query that generates the materialized view takes 120s. However, all subsequent queries that involve only changes to predicates on `l_shipdate` only take 13ms, since Afterburner only needs to query the materialized view inside the browser. Note that all of this can be done locally, without incurring any network latencies and backend server load. In contrast, without the materialized view, each subsequent query still takes 34s. We achieve "breakeven" in four queries—if we pose four variant queries, we make up for the fact that the initial backend query takes longer (even after accounting for the cost of transferring over the materialized view). Table 2 shows results for the other TPC-H queries, organized in exactly the same way. We see that in most cases, we are able to break even after just a few queries. In other words, if we wish to interactively explore the data, it makes sense to pay the cost upfront to generate the materialized view. This demonstrates the effectiveness of Afterburner's support for interactive data exploration.

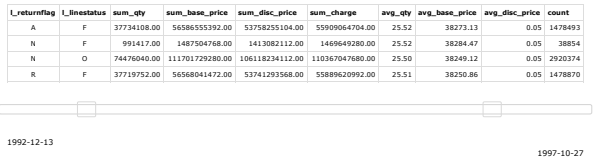
	Free column (cardinality)	FE (ms)	MV size (rows)	MV copy (ms)	BE (ms)	Break even
Q1					34,330	
Q1a	l_shipdate (2526)	13	3,817	277	119,843	4
Q2					2,556	
Q2a	p_size (50)	43	2,365,583	155,826	27,047	72
Q2b	p_type (150)	44	236,211	16,380	4,405	9
Q3					17,170	
Q3a	c_mktsegment (5)	1,147	5,662,337	107,526	19,051	8
Q3b	o_orderdate (2406)	1,280	16,553,365	268,052	36,142	18
Q4					6,924	
Q4a	o_orderdate (2526)	12	12,030	277	119,843	18
Q5					7,712	
Q5a	r_name (5)	11	25	203	8,890	2
Q5b	o_orderdate (2406)	13	12,030	517	10,125	2
Q6					4,362	
Q6a	l_shipdate (2526)	7	2,526	247	9,815	3
Q6b	l_discount (11)	6	11	375	4,590	2
Q6c	l_quantity (50)	6	50	420	8,100	2
Q7					8,388	
Q7a	l_shipdate (2526)	11	5,052	649	8,390	2
Q12					5,420	
Q12a	l_shipmode (7)	13	7	28	5,760	2
Q12b	l_receiptdate (2554)	12	4,985	163	7,219	2
Q14					4,052	
Q14a	l_shipdate (2526)	7	2,526	364	29,298	8
Q17					20,374	
Q17a	l_shipmode (7)	6	25	468	142,773	8
Q17b	l_receiptdate (2555)	7	40	678	120,131	6
Q20					8,546	
Q20a	n_name (25)	22	447,508	10,699	10,000	3
Q21					23,540	
Q21a	o_orderstatus (3)	66	96,037	2,692	30,202	2
Q21b	n_name (25)	94	999,953	16,221	76,754	4

Table 2: Time to create and copy materialized views to the frontend and the latency of subsequent frontend queries to support interactive data exploration.

The astute reader may have noticed that our entire scenario does not necessarily depend on Afterburner running in the browser. The materialized views could be stored on the backend (i.e., MonetDB), and subsequent queries can be posed against it. While true, we see two advantages of Afterburner: follow-up queries are all local and therefore do not require network overhead and incurring additional backend load. We are thus able to support truly interactive querying since it doesn't matter how busy the backend is, which might be answering queries from many users. The problem of meeting latency requirements for interactive visualizations becomes more challenging when we consider multiple users. There is a rich literature on performance isolation for shared resources, but running on the client is a simple and effective solution. We lack the space to show detailed experimental results, but an Afterburner query against the materialized view is about as fast as the same query against MonetDB on the backend. Once we factor in network latencies, Afterburner is often faster, which leads to a better user experience. From a different perspective, we argue that Afterburner represents a better solution because the materialized views are transient: if these were stored on the backend, the user would require database write privileges, which involves a whole set of administrative issues.

4. DEMONSTRATION PLAN

Our demonstration will be modeled after the evaluation and break down into two parts. First, we will demonstrate Afterburner as a standalone analytical database that runs inside the browser, both on a laptop and on a tablet. We will prepare the TPC-H scenario described in Table 1 and associated queries, along with a few other datasets.



l_returnflag	l_linestatus	sum_qty	sum_base_price	sum_disc_price	sum_charge	avg_qty	avg_base_price	avg_disc_price	count
A	F	37734108.00	56586555392.00	53758255104.00	55909064704.00	25.52	38273.13	0.05	1478493
N	F	991417.00	1487504768.00	1413082112.00	1469649280.00	25.52	38284.47	0.05	38854
N	O	74476040.00	111701729280.00	106118234112.00	110367047680.00	25.50	38249.12	0.05	2920374
R	F	37719752.00	56568041472.00	53741293568.00	55889620992.00	25.51	38250.86	0.05	1478870

1992-12-13

1997-10-27

Figure 3: An interactive visualization of TPC-H Query 1 using Afterburner.

Second, we will demonstrate Afterburner's ability to support interactive data exploration via automatically-generated materialized views. Once again, we will illustrate the system running on both a laptop and a tablet. The key feature of our demonstration is that each of the "freed" columns is tied to an interface widget that supports interactive browsing. In our running example of TPC-H Q1, follow-up queries that involve different predicates on `l_shipdate` can be translated from positions on a date slider bar, as shown in Figure 3. For each query, we are able to switch between two modes: with and without Afterburner optimizations. Without the FREE clause optimization, every new setting of the slider would involve a backend query and incur a latency of 34s. With Afterburner, the user will be able to drag the sliders and the results table above will change dynamically (with a 13ms lag). Clearly, this yields a far better interactive data exploration experience. We will prepare such visualizations for each of the queries in Table 2; for now, we have hard-coded the appropriate interface widget for each freed column. Demo visitors can pick some queries and columns to free before they completely disconnect the client (turn off the laptop's or tablet's wireless) and still be able to interact with the visualizations.

5. REFERENCES

[1] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. *SIGMOD*, pp. 1383–1394, 2015.

[2] I. T. Bowman and K. Salem. Semantic prefetching of correlated query sequences. *ICDE*, pp. 1284–1288, 2007.

[3] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. *SIGMOD*, pp. 149–160, 1996.

[4] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. *SIGMOD*, pp. 331–342, 2001.

[5] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. *VLDB*, pp. 358–369, 1995.

[6] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast queries over heterogeneous data through engine customization. *PVLDB*, 9(12):972–983, 2016.

[7] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[8] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. *VLDB*, pp. 199–210, 2006.

[9] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. *ICDE*, pp. 613–624, 2010.

[10] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI*, 2012.