SOSP: G: Xylem: dynamic, partially-stateful data-flow for high-performance Web applications

Jon Gjengset (MIT CSAIL)

Advisors: Malte Schwarzkopf, Eddie Kohler (Harvard), M. Frans Kaashoek, Robert Morris

Abstract. Xylem is a new storage backend for read-heavy web applications based on dynamic, partially-stateful data-flow. Xylem provides a web application with significantly improved read performance for commonly-executed queries by pre-computing their results and incrementally maintaining them as writes arrive. The application supplies a relational schema and SQL query definitions, which Xylem compiles into a streaming data-flow. Application writes stream through the data-flow to update stateful caches of the queries' results, which serve reads efficiently.

Xylem makes two principal contributions to data-flow systems. First, Xylem supports partially-stateful data-flow, which "backfills" missing state on demand. This avoids the need for time-windowed state and queries in current systems. Second, Xylem allows the data-flow to change without restarting, adding sub-graphs that compute new expressions while reusing existing state and computation wherever possible. This enables applications to change their query set over time, without having to restart the Xylem backend and re-process old data.

Our prototype implementation of Xylem scales several orders of magnitude further than MySQL for a query common in a real web application workload. This increased scalability also translates into higher efficiency; Xylem handles the same load as MySQL using significantly fewer computational resources.

1. Problem and motivation

Web applications must serve many users at low latency, composing structured data from a persistent store for each response. The vast majority of store accesses are reads, which makes repeated evaluation of read queries over the normalized schema of a relational database inefficient [18, 19]. To keep up with high loads, applications therefore often either optimize the schema for better read performance (*e.g.*, "denormalizing" it by storing and updating computed values explicitly), or cache the query results in a separate key-value store [4, 18]. While these techniques increase read performance, the application must now also maintain these auxiliary values when it writes to the backing store. This forces a choice between slow, but convenient, relational queries and fast, but hard-to-manage, application-side caches.

However, this choice is not fundamental. Since the database knows the queries, it has enough information to keep cached query results up-to-date without involving the application. This is the idea behind *incremental materialized views*; applications prepare long-lived queries with the database, and the database efficiently maintains their results as writes arrive. When the application issues a read, the query results are then immediately available.

	HotCRP		Lobste.rs
Query type	**queries**	**time**	**queries**
read (`SELECT`)	96.78%	87.62%	88.4%
write (`INSERT`, `UPDATE`)	2.51%	7.41%	7.6%
schema change	0.18%	4.84%	–
other (locking, `SHOW`)	0.53%	0.14%	4.0%

Table 1: The query distributions of a month of HotCRP (23k sampled queries) and Lobste.rs activity (50M queries, 2M requests) [3] skew heavily towards reads.

Streaming data-flow systems provide an attractive platform for such incremental materialization: they are inherently incremental, and can easily be distributed across CPUs and physical machines for scale-out performance.[5, 13–15, 22] An application whose queries are encoded in a data-flow computation feeds its writes into the data-flow system, and performs reads by inspecting the systems' outputs. But, existing data-flow systems are missing important features needed to support web applications. First, it is prohibitively expensive to store and maintain the full results of all application queries, so the ability to keep only "useful" query results cached is critical. That is, they do not support *partial materialization*. Second, the queries that an application executes changes over time [8, 20], but current data-flow systems must throw away all cached state any time the set of queries changes, and re-do any computation performed thus far. That is, they do not allow *query evolution*.

In this paper, we describe Xylem, a data-flow system that solves these intertwined issues and provides incremental, dynamic view maintenance with partial materialization geared towards read-heavy web applications.

2. Background and related work

Web applications issue complex queries that involve substantial computation to render any given page, but those queries are overwhelmingly reads. Table 1 illustrates this with empirical data from an active HotCRP[1] site and the pro-

[1] `https://hotcrp.com/`

duction deployment of Lobste.rs[2], a tech news aggregation site. Over 95% of the queries these applications issue are SELECT queries, and such queries consume 88% of the total query execution time in HotCRP. In Lobste.rs, the most frequently accessed page (an individual story) generates 16 queries, of which 15 are reads. The performance of these queries is sufficiently important that application developers go through significant effort to optimize them. For example, Lobste.rs stores both individual votes for stories *and* each story's computed vote count. This allows retrieval of the most-upvoted stories without dynamically computing the counts, albeit at the cost of "de-normalizing" the relational schema and making updates more complex. As another example of application-level effort to optimize queries, Lobste.rs often batches multiple reads using `IN()` over a list of keys (*e.g.*, stories on the front page).

Web sites often deploy an **in-memory key-value cache** (like Redis or memcached) to speed up common-case read queries. Such a cache allows efficiently fetching the results of a query when the underlying records are unchanged, although those results are now more likely to be stale. But, to keep the cache up to date as the DB changes, the application must implement cache invalidation and replacement policies. This process is error-prone, and introduces further application-side complexity to mitigate performance problems like "thundering herds" [18, 19].

The database literature refers to keeping a cache of query results up to date as **materialized view maintenance**, where a *view* is a query that is known in advance. In particular, state-of-the-art research databases support *incremental* view maintenance, wherein just the *changes* to query results in response to writes are computed. They use repeated execution of "delta queries", derivatives of SQL queries, to advance the materialized view over time [1, 12, 17]. However, these systems primarily target static, batched workloads, not the kind of interactive workloads that characterize web applications.

When applications want evolving query results over a long-running stream of updates, such as for interactive analytics dashboards, they often turn to **stream-processing systems** [6, 13]. These systems use streaming data-flow abstractions to incrementally update results, and efficiently process large amounts of data quickly by distributing the computation across cores and computers. One major drawback of these systems is that they must maintain *state* at some internal data-flow operators (*e.g.*, joins and aggregations), and this state could grow without bound. To mitigate this, existing systems "window" their state by limiting it either to a fixed number of records (*e.g.*, the most recent 1k records), or a fixed time window (*e.g.*, data for a sliding five-minute window). This windowing makes it impossible to process the general queries needed for Web sites using streaming data-flow, as they often need to return old as well as recent state. Moreover, stream processors are not nearly as flexible as a database that can execute any relational query on its schema: once active, the data-flow computation generally cannot change, precluding dynamically-generated queries and application upgrades without restarting the data-flow.

[2] `https://lobste.rs`

3. The Xylem approach

Xylem combines the persistent store, the caching layer, and elements of the application logic in a single, dynamically changing *data-flow computation* that serves as the Web application backend. Each write streams through the data-flow computation generated from the current queries, and incrementally updates any relevant stored query results. Whenever the application changes the set of queries, Xylem adapts the data-flow graph to include new queries without interruption in client service, and reuses existing state wherever possible. Xylem's design draws on existing data-flow systems for processing of "big data" [10, 16, 23] and for streaming analytics [5, 13–15, 22], but departs from these prior systems in several key aspects. Its non-windowed, partial state, in particular, is unlike the abstractions stream-processing systems commonly provide [6, 13, 14], and makes Xylem similar to materialized view maintenance systems (*e.g.*, DBToaster [1], Pequod [11]).

Xylem implements incremental view maintenance using dynamic data-flow with support for partial materialization. This allows Xylem to support the fast reads of key-value caches, the efficient updates and parallelism of streaming data-flow, and, like a classic database, enables flexibly changing the queries and base table schemas without downtime. Xylem makes this possible through two following two data-flow innovations:

1. it introduces *ancestor queries* in a data-flow graph, which in turn enable partial materialization and eviction; two features necessary to support materialized views over large datasets without windowed state; and
2. it supports low-overhead, *dynamic modification* of its data-flow graph through multi-query optimization to support changes in query sets and base table schemas, without any downtime for reads, minimal downtime for writes, and efficient re-use of existing state.

3.1 Programming interface

The application provides Xylem with a data-flow specification written as a SQL-like program. It includes (*i*) base table definitions, (*ii*) SQL expressions that define intermediate named views that can be re-used, and (*iii*) SQL expressions defining "leaf" views that applications read from. Listing 1 shows an example specification for a Lobste.rs-like news aggregator application.

Figure 2 shows Xylem's interface. Queries that are parameterized can be executed using `lookup`; for example, the `ArticleWithVC` view is keyed by an article's id, so `lookup(AWVC, 1)` would fetch the title and vote count of article 1. For queries without parameters, `scan` returns all rows. Clients provide full records to `insert` and primary

```
/* base tables */
CREATE TABLE Article (id int, title text);
CREATE TABLE Vote (user int, article int);
/* intermediate view */
VoteCount:     /* count votes per article */
  SELECT article, COUNT(*) AS votes
    FROM Vote GROUP BY article;
/* leaf views */
VIEW ArticleWithVC: /* full article details */
  SELECT id, title, votes
    FROM Article
    JOIN VoteCount ON VoteCount.article = Article.id
   WHERE Article.id = ?;
```

Listing 1: Xylem program for a Lobste.rs-style news aggregator where users vote for articles.

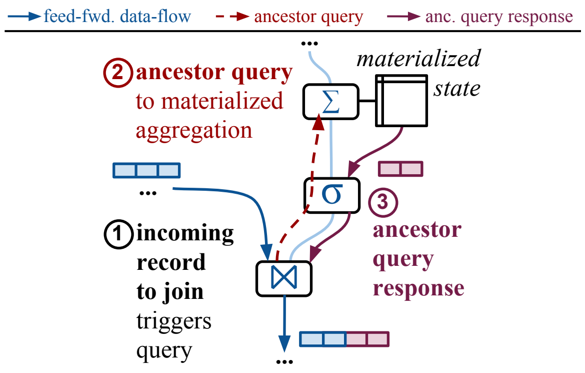
feed-fwd. data-flow
ancestor query
anc. query response
materialized state
2 ancestor query to materialized aggregation
Σ
σ
⋈
1 incoming record to join triggers query
3 ancestor query response

Figure 1: A join queries its ancestor to find matching records.

```
# read
lookup(leaf view, parameter[]) -> record[]
scan(leaf view) -> record[]
# write
insert(table, record)
delete(table, key)
# data-flow change
migrate_to(sql_expr)
```

Listing 2: Application-facing Xylem interface.

keys to `delete`. To provide compatibility with existing applications that issue ad-hoc SQL queries, Xylem implements the MySQL binary protocol, and maps prepared statements into lookups on parameterized Xylem views automatically.

Finally, the API provides a `migrate_to` call that changes the data-flow specification using SQL snippets like Listing 1. An application may change the specification to add a new view; to change views during an upgrade; or to change the input schema. Xylem expects such changes to be common — *e.g.*, triggered several times a day by continuous deployment [8, 20] — and aims to complete them quickly.

3.2 Data-flow write processing

The Xylem data-flow graph is a directed acyclic graph of relational operators such as aggregations, joins, and projections, generated from the SQL specification. When a write arrives, Xylem appends it to a durable base table and injects it into the data-flow graph as an *update* (a collection of changes to records, initially a singleton). The write percolates through the operators of the data-flow graph; each operator processes the update, computes the corresponding results, stores them in its state if appropriate, and forwards the resulting new updates to any children.

Xylem operators fall into three categories: stateless, stateful, and joins. Stateless operators (*e.g.*, filters, projections) process records without context, and need no special consideration. Stateful operators (*e.g.*, count, min/max, Top-k) must keep track of their current state to incrementally maintain their results. Joins are special as they need to query their ancestors using the join key to produce joined output records. In Xylem, such *ancestor queries* are sent to the closest stateful operator — reaching through stateless operators on the way — and produce a stream of records matching the join key (Figure 1). This is unlike the joins in existing data-flow systems, which internally keep a windowed materialization of their inputs.

Xylem keeps at least one index for each materialization of state. It chooses index columns using *indexing obligations* derived by analysing the data-flow; each state has indexes required by the semantics of the attached operator (*e.g.*, aggregations have indices on group keys), and indexes for the ancestor queries it may receive from downstream operators.

3.3 Defining correctness

The state of Xylem's data-flow operators and the contents of the materialized views at the leaves of its data-flow graph are *eventually consistent*. In other words, a write inserted at an input (*i.e.*, a base table) may take some time to become visible, and may appear in different states at different times. Eventual consistency is attractive for performance (*e.g.*, because views need not synchronize to expose updates) and scalability (*e.g.*, because no total ordering of writes is required); it is also sufficient for many Web applications [7, 18, 21]. However, despite its weak consistency guarantee, Xylem must nevertheless eventually compute accurate results, which requires some care.

Xylem, like other data-flow systems [2, 15], guarantees **exactly-once updates**: every update that affects an operator's state is applied to it exactly once. This means that Xylem cannot, for example, apply a write to `Vote` to the state of the `VoteCount` aggregation operator several times and render the count permanently inaccurate. Great care must be taken to ensure that this property holds internally in the Xylem data-flow graph, especially in the face of forks, joins, and the ancestor queries used by partial materialization.

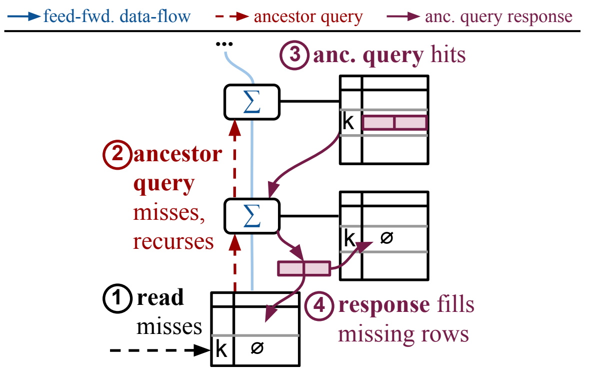

Figure 2: Operators in the Xylem data-flow graph use ancestor queries to query into upstream materializations.

3.4 Partial materialization

Partial materialization is a well-known technique in database materialized views [25, 26], but is novel to data-flow systems, which currently either materialize all state or time-window their state. Data-flow systems struggle with partial materialization because they are primarily *feed-forward*: writes traverse the computation graph from the inputs to the outputs along uni-directional edges. But if a lookup misses (*i.e.*, the state for a requested key is not available), that miss must be communicated back up the graph, in the opposite direction of the data-flow. This is not possible in existing data-flow systems, but it is in Xylem's materializations.

Xylem supports this form of backwards control flow through *ancestor queries* (Figure 2). When an operator needs state in order to perform a computation (*e.g.*, a join needs to do a lookup for the join key against the other side of the join), it issues an ancestor query up the graph to the ancestor operator whose output it wants to query. If the operator has the state necessary to satisfy the query, it sends that state forward to the requesting operator as a *backfill*. If it does not, it issues a second ancestor query to *its* ancestor(s) as necessary to generate the missing state. That ancestor treats the backfill request as if it got an ancestor query, recursing if necessary. This process continues until all the needed state has been fed forward through the graph all the way to the initiating operator. Any operators along the backfill path store the computed intermediate results so that any subsequent ancestor queries for the same key do not need to recurse.

As records can be backfilled if necessary, operators are generally free to evict stored query results at their own discretion. This could be when they believe they are no longer necessary (*e.g.*, an article that has fallen off the front page of a news site, and is thus no longer as popular), or to alleviate memory pressure.

It is important that Xylem sends backfills along the regular data-flow forward path, in-line with regular application writes, without re-ordering. If Xylem did not do this, regular writes that happen to be present in a backfill could arrive *after* that backfill at the requesting operator, causing those writes to take effect twice.

Ensuring exactly-once. Since ancestor queries can cause blocking recursive backfills, Xylem does not perform them on the forward data-processing path; instead, it discards feed-forward updates that touch a key that is found to be missing. However, this introduces an important invariant that Xylem must take care to maintain: an ancestor cannot evict state for a key that is still present in a child. Otherwise, evicting a key on an operator on one side of a join could cause writes on the other side to be discarded as the ancestor query would now miss. This in turn means that downstream materializations which have not evicted the same key are not receiving relevant writes. Xylem maintains this invariant by always evicting a key from an entire subgraph below a given node, and forcing evictions when ancestor queries miss on the forward processing path.

3.5 Query evolution

The queries of web applications change over time as existing features are changed or removed, and as new features are added. However, existing data-flow systems do not allow changes to the running computation except by shutting the system down and starting afresh. For an application that has amassed data over a long time, the cost of re-processing all of its data, and effectively re-populating all of its caches, whenever the application queries changes is not acceptable.

Xylem supports dynamically modifying the running data-flow computation without discarding existing cached state, and with minimal disruption to the running system. When the application calls `migrate_to` to initiate a change to its query set, Xylem first construct a new *joint query plan* (*i.e.*, a graph of data-flow operators) for that query set. It then identifies places where the new plan overlaps with the currently running plan using multi-query optimization (MQO) [9, 24], which tries to maximize sharing across a batch of expressions, with the freedom to rewrite expressions to increase matches. Xylem also identifies cases where base table schemas have changed, and automatically introduces operators to ensure that old and new writes can co-exist wherever possible.

To apply the necessary changes identified by MQO, Xylem first appends all new operator nodes to the running data-flow graph. These new operators are then connected to their ancestors in the graph. Stateless and partially materialized operators can begin processing reads and writes immediately after they are added, and will perform ancestor queries as necessary to populate their state. Operators that require full materialization (*e.g.*, global aggregations that have no key) initially ignore all arriving writes, and immediately issue an ancestor query *without a key* to their ancestor(s), effectively asking them to replay *all* of their state. This latter process is expensive, and holds up processing of writes and ancestor queries in the ancestor, but are not much more expensive

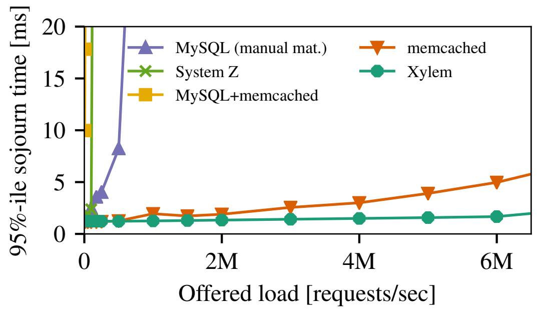

Figure 3: Xylem outperforms alternative approaches for a read-heavy workload on the `ArticleWithVC` query, and scales to >6M requests/sec.

than the table scan a relational database would perform given a similar query. Anecdotally, across the schema and query set changes the HotCRP conference management system has gone through since 2006, upwards of 95% of changes introduces only partial or stateless operators.

4. Results and Contributions

Our Xylem prototype is implemented in 40k lines of Rust, and includes a MySQL binary protocol adapter that exposes a standard MySQL server API to unmodified applications. The adapter translates prepared statements and ad-hoc queries into migrations on Xylem's data-flow, and applies reads and writes using Xylem's API when executing a statement or ad-hoc query. Our prototype also supports parallelism via sharding and distribution of shards across different machines.

Comparison with alternatives. We consider Xylem's performance for a specific, highly used Lobste.rs query (`ArticleWithVC` in Listing 1) compared to several alternatives. We first populate with 100k articles; during measurement, clients read the vote count of different articles, and insert votes for random stories following a Zipfian distribution ($s = 1.08$).

We compare (*i*) Lobste.rs's current manual materialization of vote counts in a MySQL column; (*ii*) an incrementally-maintained `ArticleWithVC` materialized view in "System Z", a widely-used, commercial database; (*iii*) a demand-filled memcached cache [18] in front of MySQL; (*iv*) memcached storing vote counts *only* in memory; and (*v*) Xylem. Each server runs on one 16-core VM, and a large pool of open-loop clients run on a cluster of different 16-core VM. All servers are configured to operate entirely in-memory, and with the weakest possible consistency model. In this setting, an ideal system would show as a horizontal line with low latency; in reality, each system displays a vertical "hockey stick" once it fails to keep up with the offered load.

Figure 3 illustrates that neither System Z's materialized view nor the MySQL+memcached setup perform well: the former struggles with write contention, and the latter experiences the "thundering herd" problem [18, §3.2.1] The strategy used in the real Lobste.rs — materializing the vote count in MySQL — does better, but fails to scale beyond 800k requests/sec. Xylem, by contrast, outperforms even the memcached-only setup — even though memcached neither has durable storage nor a relational query interface. This is because Xylem can satisfy the large majority of requests directly from its materialized views, and thus does needs to do little work to satisfy the application reads. Xylem outperforms memcached because it uses a lock-free hash table for its materialized leaf views, while memcached uses fine-grained locks that become contended under skewed load.

Other workload mixes, *e.g.*, a uniform popularity distributions or a 50% read/write workload, show similar results. Xylem also efficiently supports nearly all other queries issued by the real Lobste.rs web application without modifications, and delivers similar performance improvements on all of them[3]. Taken together, these results demonstrate that the Xylem design indeed provides high performance for realistic web application workloads with minimally intrusive changes.

Availability. Xylem is open-source and available at `https://github.com/mit-pdos/distributary`.

References

[1] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. "DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views". In: *Proceedings of the VLDB Endowment* 5.10 (June 2012), pp. 968–979.

[2] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, et al. "MillWheel: Fault-tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pp. 1033–1044.

[3] Anonymized. *Lobste.rs access pattern statistics for research purposes*. Public data, but reveals authors' identity. URL provided to PC chairs. Mar. 2018. (Visited on 03/12/2018).

[4] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, et al. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *Proceedings of the USENIX Annual Technical Conference*. San Jose, California, USA, June 2013, pp. 49–60.

[5] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. "Apache Flink: Stream and batch processing in a single engine". In: *IEEE Data Engineering* 38.4 (Dec. 2015).

[3] Without the MySQL binary protocol shim, Xylem delivers lower latencies, and can support a wider range of queries.

[6] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, et al. “Realtime Data Processing at Facebook”. In: *Proceedings of the 2016 SIGMOD International Conference on Management of Data*. San Francisco, California, USA, 2016, pp. 1087–1098.

[7] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, et al. “PNUTS: Yahoo!’s Hosted Data Serving Platform”. In: *Proceedings of the VLDB Endowment* 1.2 (Aug. 2008), pp. 1277–1288.

[8] Dror G. Feitelson, Eitan Frachtenberg, and Kent L. Beck. “Development and Deployment at Facebook”. In: *IEEE Internet Computing* 17.4 (July 2013), pp. 8–17.

[9] Sheldon Finkelstein. “Common Expression Analysis in Database Applications”. In: *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data*. Orlando, Florida, USA, June 1982, pp. 235–245.

[10] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. “Dryad: Distributed Data-parallel Programs from Sequential Building Blocks”. In: *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys)*. Lisbon, Portugal, Mar. 2007, pp. 59–72.

[11] Bryan Kate, Eddie Kohler, Michael S. Kester, Neha Narula, Yandong Mao, and Robert Morris. “Easy Freshness with Pequod Cache Joins”. In: *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Seattle, Washington, USA, Apr. 2014, pp. 415–428.

[12] Christoph Koch, Daniel Lupei, and Val Tannen. “Incremental View Maintenance For Collection Programming”. In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS ’16. San Francisco, California, USA: ACM, 2016, pp. 75–90.

[13] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, et al. “Twitter Heron: Stream Processing at Scale”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. Melbourne, Victoria, Australia, May 2015, pp. 239–250.

[14] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. “Differential dataflow”. In: *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, California, USA, Jan. 2013.

[15] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. “Naiad: A Timely Dataflow System”. In: *Proceedings of SOSP*. Nov. 2013, pp. 439–455.

[16] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. “CIEL: a universal execution engine for distributed data-flow computing”. In: *Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Mar. 2011, pp. 113–126.

[17] Milos Nikolic, Mohammad Dashti, and Christoph Koch. “How to Win a Hot Dog Eating Contest: Distributed Incremental View Maintenance with Batch Updates”. In: *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. San Francisco, California, USA, 2016, pp. 511–526.

[18] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, et al. “Scaling Memcache at Facebook”. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Lombard, Illinois, USA, Apr. 2013, pp. 385–398.

[19] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. “Transactional Consistency and Automatic Management in an Application Data Cache”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, BC, Canada, 2010, pp. 279–292.

[20] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. “Continuous Deployment at Facebook and OANDA”. In: *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. Austin, Texas, USA, 2016, pp. 21–30.

[21] Werner Vogels. “Eventually Consistent”. In: *Communications of the ACM* 52.1 (Jan. 2009), pp. 40–44.

[22] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. “Discretized Streams: Fault-tolerant Streaming Computation at Scale”. In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, Pennsylvania, USA, Nov. 2013, pp. 423–438.

[23] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, et al. “Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing”. In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, Apr. 2012, pp. 15–28.

[24] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. "Efficient Exploitation of Similar Subexpressions for Query Processing". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Beijing, China, 2007, pp. 533–544.

[25] Jingren Zhou, Per-Åke Larson, and Jonathan Goldstein. *Partially Materialized Views*. Tech. rep. MSR-TR-2005-77. Microsoft Research, June 2005.

[26] Jingren Zhou, Per-Åke Larson, Jonathan Goldstein, and Luping Ding. "Dynamic Materialized Views". In: *Proceedings of the 23rd International Conference on Data Engineering (ICDE)*. Istanbul, Turkey, Apr. 2007, pp. 526–535.