Announcing incremental MapReduce for VoltDB - explaining materialized views

We are sometimes asked, “When will you add incremental MapReduce to VoltDB?”, a feature some NoSQL systems have added. For an operational system like ours, the fit is obvious: the system pays a small fixed cost on update. Then, aggregations, counts and filtering on huge datasets all can be done in an instant. We did, in fact, implement this feature - in version 1.0 - with our materialized views. At the time, we didn’t have the foresight to call it “Incremental MapReduce,” but rather used the name “Materialized Views.” So when someone suggests we add Incremental MapReduce, we tell them about our materialized views.

Oddly enough, in 2015, for some people, it’s easier to wrap their heads around Incremental MapReduce than our materialized views.

So let’s talk about materialized views for a minute.

- What are VoltDB materialized views?
- How is data in materialized views accessed?
- How do streaming analytics support per-event transactions?

What are VoltDB Materialized Views?

A materialized view is an aggregation of data in the database that is calculated and persisted (materialized) into a new table. For example, imagine a table of US voters. Each record has a voter's name, zip code, and registration status. A materialized view can be declared in DDL to keep a real-time running subtotal of registered voters by zip code as voters are added or removed from the database or change addresses.

```
1.  CREATE TABLE voters (
2.      name varchar(50),
3.      zipcode varchar(5),
4.      registration smallint
5.  );
6.
7.  CREATE VIEW registrations_by_zipcode (
8.      zipcode, registered_voters
9.  ) AS
10. SELECT zipcode, count(*) from voters where registration=1 GROUP BY zipcode;
```

Example Voters table:

name	zipcode	registration
Abbey	01776	1
Billy	01730	0
Chris	01701	1
David	01776	1

The resulting registrations_by_zipcode view:

zipcode	registered_voters
01776	2
01701	1

In VoltDB, you can add materialized views to tables that change very rapidly to maintain streaming aggregations and summaries of real-time state. VoltDB keeps materialized views in-memory and fully up-to-date with the current transaction. VoltDB materialized views are "live" and do not need to be periodically refreshed or rebuilt. Each completed transaction updates the materialized view automatically. Consequently, materialized views in VoltDB are always transactionally consistent with the base table. The system maintains this invariant without user intervention or external commands.

In almost all interesting cases, materialized views are derived from a partitioned table. Consequently, the views are also partitioned across a cluster and gain the advantages of VoltDB's horizontal scale-out and performance. Materialized views are highly available and active/active replicated like all user data in a VoltDB cluster.

VoltDB currently supports materialized views derived from a single base table. Here's the supported syntax excerpted from the online documentation.

```
CREATE VIEW view-name ( view-column-name [,...] )
AS SELECT { column-name | selection-expression } [AS alias] [,...]
FROM table-name
[WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
GROUP BY { column-name | selection-expression } [,…]
```

How are Materialized Views Accessed?

Materialized Views on partitioned tables follow the same rules as other partitioned tables in terms of query support. We support intra-partition joins that include an equality expression on the partitioning column. Views that contain the partitioning column in the GROUP BY predicate can be joined to other views and to other partitioned tables accordingly.

Additionally, many applications use multi-partition (cluster-wide) SELECT queries to roll up results based on partitioned views. Distributed reads of view data are fast. These reads are serializable, as is all SQL in VoltDB. This pattern uses VoltDB's partitioned scaling model to scale to high-speed, write-oriented applications while supporting global reads of real-time aggregates for analytics, dashboards, BI, or other external applications.

Combining the streaming aggregation of materialized views with rich SQL support and support for consistent, ACID reads across the entire cluster opens up a broad set of use cases. One common pattern is to use views to aggregate high speed event streams by second-of-day and then use SQL queries to further aggregate by minutes, hours, or other units. This allows fast roll-ups of time-series data at different levels of granularity.

Here's an example excerpted from the windowing application included in the VoltDB download kit that calculates an average from a user-specified time range:

```
CREATE TABLE timedata
(
  uuid VARCHAR(36) NOT NULL,
  val BIGINT NOT NULL,
  update_ts TIMESTAMP NOT NULL
);

CREATE VIEW agg_by_second
(
  second_ts,
  count_values,
  sum_values
)
AS SELECT TRUNCATE(SECOND, update_ts), COUNT(*), SUM(val)
   FROM timedata
   GROUP BY TRUNCATE(SECOND, update_ts);

-- Find the average value over all tuples across all partitions for the last
-- N seconds, where N is a parameter the user supplies.
CREATE PROCEDURE Average AS
    SELECT SUM(sum_values) / SUM(count_values)
    FROM agg_by_second
    WHERE second_ts >= TO_TIMESTAMP(SECOND, SINCE_EPOCH(SECOND, NOW) - ?);
```

VoltDB SQL supports functions to convert timestamps to SECOND, MINUTE, HOUR, DAY, DAYOF-MONTH, DAYOFYEAR, MONTH, WEEK, WEEKOFYEAR, WEEKDAY, and YEAR, all of which can be mix-and-matched with view definitions and queries.

Since views scale just like the rest of VoltDB, you can distribute the cost of calculating summaries across the fast incoming write/update workload and make it very cheap to rapidly read summaries of state. In the example above, if tuples are being inserted at a rate of 15k/sec and there are four partitions, then to compute the average for the last ten seconds, VoltDB would need to scan 150k rows. Using a view, it needs to scan 1 row per second times the number of partitions, or 40 rows. There's a tremendous advantage in pre-aggregating the table sums and counts by second.

How do you combine streaming analytics with per-event decisions?

Since materialized views are always up to date, are always transactionally consistent, and are accessible via standard SQL, it is easy to use aggregations maintained in views when running transactions. Our Voter example does exactly this. The example maintains a view that groups incoming records by phone_number. The transaction that enforces a per-phone-number voting limit uses the view to lookup the previous calls registered by the incoming phone-number. A more sophisticated example could use the time-series functions discussed above to enforce a max-accesses per-minute for high-speed quota enforcement.

Let's use the Voter example to show runtime benefits of amortizing the aggregation cost across the scalable ingest workload. The example includes a heat-map that graphs which contestant leads the voting in each state. Querying the data for this heat-map requires counting votes by state, of course. Using a view, this can be done in sub-millisecond times (it requires reading 50 rows at each partition for each of the 6 contestants). Without a view, a scan of all votes is necessary and requires considerable execution time (and requires reading millions rows). Below are some concrete numbers showing execution times for roughly 10GB of data (~130M rows) per-partition on a recent Intel core i7 CPU.

Without the view, all 130M rows are scanned. Execution time is ~8 seconds:

```
1.  26> select state, count(*) from votes where state='MA' group by state;
2.  STATE  C2
3.  ------ --------
4.  MA       3,839,860
5.
6.  (Returned 1 rows in 8.13s)
```

With the view, execution time is sub-millisecond:

```
1.  31> select state, sum(num_votes) from v_votes_by_contestant_number_state where
2.  STATE  C2
3.  ------ --------
4.  MA       3,839,860
5.
6.  (Returned 1 rows in 0.00s)
```

The view implementation has some impact on the ingest workload, but even with the view, the database can process more than 40k requests per second per core. The view overhead can be compensated for with horizontal scalability - the work required can be scaled across cores and across servers in a cluster. This trade-off enables the use of real time aggregation in per-event transactions at high throughput.

```
1.  1 partition throughput with v_votes_by_contestant_number_state view:
2.  Throughput 42252/s, Aborts/Failures 0/0
3.  Throughput 45407/s, Aborts/Failures 0/0
4.  Throughput 46352/s, Aborts/Failures 0/0
5.  Throughput 44341/s, Aborts/Failures 0/0
6.
7.  1 partition throughput without v_votes_by_contestant_number_state_view:
8.  Throughput 50525/s, Aborts/Failures 0/0
9.  Throughput 52503/s, Aborts/Failures 0/0
10. Throughput 50961/s, Aborts/Failures 0/0
11. Throughput 51190/s, Aborts/Failures 0/0
```

Conclusion

VoltDB is an in-memory, ACID, scale-out, SQL database. It can process tens of thousands to millions of transactions per second. It is also capable of real-time SQL analytics against high-velocity data streams. An important part of VoltDB's real-time analytics capabilities is built around its materialized view implementation. VoltDB materialized views support real time aggregation and summary and support combining real-time analytics with per-event decisions.

[1]: http://docs.voltdb.com/UsingVoltDB/ddlref_createview.php