Couchbase Analytics: NoETL for Scalable NoSQL Data Analysis

Murtadha Al Hubail[1] Ali Alsuliman[1] Michael Blow[1]
Michael Carey[12] Dmitry Lychagin[1] Ian Maxon[12] Till Westmann[1]
[1]Couchbase, Inc. [2]University of California, Irvine
mike.carey@couchbase.com

ABSTRACT

Couchbase Server is a highly scalable document-oriented database management system. With a shared-nothing architecture, it exposes a fast key-value store with a managed cache for sub-millisecond data operations, indexing for fast queries, and a powerful query engine for executing declarative SQL-like queries. Its Query Service debuted several years ago and supports high volumes of low-latency queries and updates for JSON documents. Its recently introduced Analytics Service complements the Query Service. Couchbase Analytics, the focus of this paper, supports complex analytical queries (e.g., ad hoc joins and aggregations) over large collections of JSON documents. This paper describes the Analytics Service from the outside in, including its user model, its SQL++ based query language, and its MPP-based storage and query processing architecture. It also briefly touches on the relationship of Couchbase Analytics to Apache AsterixDB, the open source Big Data management system at the core of Couchbase Analytics.

PVLDB Reference Format:
Murtadha Al Hubail, Ali Alsuliman, Michael Blow, Michael Carey, Dmitry Lychagin, Ian Maxon, and Till Westmann. Couchbase Analytics: NoETL for Scalable NoSQL Data Analysis. *PVLDB*, 12(12): 2275-2286, 2019.
DOI: https://doi.org/10.14778/3352063.3352143

1. INTRODUCTION

It has been fifty years (yes, a full half-century!) since Ted Codd changed the face of data management with the introduction of the relational data model [12, 13]. His simple tabular view of data, related by values instead of by pointers, made it possible to design declarative query languages to allow business application developers and business analysts to interact with their databases logically rather than physically – specifying what, not how, they want. The ensuing decades of research and industrial development brought numerous innovations, including SQL, indexing, query optimization, parallel query processing, data warehouses, and many other features that are now taken for granted in today's multibillion-dollar database industry. The world of data and applications has changed, however, since the early days of mainframes and their attached terminals.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352143

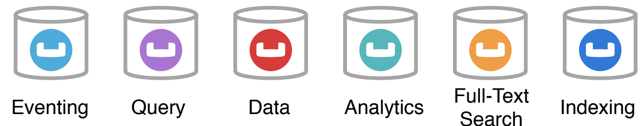


Figure 1: Couchbase Server Overview

Today's mission-critical applications demand support for millions of interactions with end-users via the Web and mobile devices. In contrast, traditional database systems were built for thousands of users. Designed for strict consistency and data control, they tend to lack agility, flexibility, and scalability. To handle a variety of use cases, organizations today often end up deploying multiple types of databases, resulting in a "database sprawl" that brings with it inefficiencies, slow times to market, poor customer experiences, and for analytics, slow times to insight as well. This is where Couchbase Server comes in – it aims to reduce the degree of database sprawl and to reduce the mismatch between the applications' view of data and its persisted view, thereby enabling the players – from application developers to DBAs and now data analysts as well – to work with their data in its natural form. It adds the much-needed agility, flexibility, and scalability while also retaining the benefits of a logical view of data and SQL-like declarative queries.

2. COUCHBASE SERVER

Couchbase as a company was created through a merger of two startups, Membase and CouchOne, in 2011. This brought two technologies together: Membase Server, a distributed key-value database based on memcached, and CouchDB, a single-node document database supporting JSON. The goal of the merged company was to combine and build on the strengths of these two complementary technologies and teams. Couchbase Server today is a highly scalable document-oriented database management system [5]. With a shared-nothing architecture, Couchbase Server exposes a fast key-value store with a managed cache for sub-millisecond data operations, secondary indexing for fast queries, and a high-performance query engine (actually two complementary engines) for executing declarative SQL-like N1QL* queries.

Figure 1 provides a high-level overview of Couchbase Server. (Not included in the figure is Couchbase Mobile, which extends

*Short for Non-1NF Query Language.

Couchbase data platform to the edge, securely managing and syncing data from any cloud to all edge devices.) Architecturally, the system is organized as a set of services that are deployed and managed as a whole on a Couchbase Server cluster. Physically, a cluster consists of a group of interchangeable nodes that operate in a peer-to-peer topology and the services running on each node can be managed as required. Nodes can be added or removed through a rebalance process that redistributes the data evenly across all nodes. The addition or removal of nodes can also be used to increase or decrease the CPU, memory, disk, or network capacity of a cluster. This process is done online and requires no system downtime. The ability to dynamically scale the cluster capacity and individually map services to sets of nodes is sometimes referred to as Multi-Dimensional Scaling (MDS).

An important aspect of the Couchbase Server architecture is how data mutations are communicated across services. Mutation coordination is done via an internal Couchbase Server protocol, called the Database Change Protocol (DCP), that keeps components in sync by notifying them of all mutations to documents managed by the Data Service. The Data Service in Figure 1 is the producer of such mutation notifications, with the other services participating as DCP listeners.

The Couchbase Data Service provides the foundation for document management. It is based on a memory-first, asynchronous architecture that provides the core capabilities of caching, data persistence, and inter-node replication. The document data model for Couchbase Server is JSON [17], a flexible, self-describing data format capable of representing rich structures and relationships. Documents are stored in containers called buckets. A bucket is a logical collection of related documents in Couchbase, similar to a database or a schema in a relational database. It is a unique key space and documents can be accessed using a (user-provided) document ID much as one would use a primary key for lookups in an RDBMS.

Unlike a traditional RDBMS, the "schema" for documents in Couchbase Server is a logical construct defined based on the application code and captured in the structure of the stored documents. Since there is no explicitly defined schema to be maintained, developers can add new objects and properties at any time simply by deploying new application code that stores new JSON data without having to also make and deploy corresponding changes to a static schema. This provides significant agility, allowing modern applications to evolve quickly and seamlessly.

The Indexing, Full-Text Search, and Query Services shown in Figure 1 coordinate through DCP to provide highly performant, user-facing document database management functionality, supporting high volumes of low-latency queries and updates for JSON documents. The Indexing Service provides scalable global secondary indexing for the data managed by the Data Service, and the Full-Text Search service extends Couchbase Server's indexing capabilities to include rich text indexing and search (plus a set of APIs for search-oriented applications that prefer to interact with this service directly). The Query Service ties this all together by exposing Couchbase Server's database functionality through N1QL, a declarative, SQL-based query language that relaxes the rigid 1NF and strongly-typed schema demands of the relational SQL language standard.

Two recently introduced services also appear in Figure 1. One is the Eventing Service, a framework that application developers can use to respond to data changes in a Couchbase cluster in real time. The Eventing Service provides an Event-Condition-Action based model for invoking functions registered by applications. Last but not least, particularly for this paper, is the Analytics Service, which was first introduced in Couchbase Server 6.0 in late 2018.

3. COUCHBASE ANALYTICS SERVICE

The Couchbase Analytics Service complements the Query Service by providing support for complex and potentially expensive ad-hoc analytical queries (e.g., large joins and aggregations) over collections of JSON documents.[†] Figure 2 provides a high-level illustration of the role that the Analytics Service plays in Couchbase Server. The Data Service and the Query Service provide user-facing applications with low-latency key-value and/or query-based access to their data. The design point for these services is a large number of users making relatively small, inexpensive requests. In contrast, the Analytics Service is designed to support a much smaller number of users posing much larger, more expensive N1QL queries against a real-time shadowed copy of the same JSON data.

The technical objectives of Figure 2's approach are similar to those of Gartner's HTAP concept for hybrid workloads [15]: Data doesn't have to be copied from operational databases to data warehouses before analysis can commence, operational data is readily available for analytics as soon as it is created, and analyses always utilize fresh application data. This minimizes the time to insight for application and data analysts by allowing them to immediately pose questions against their operational data in terms of its natural data model. This is in contrast to traditional architectures [11] that, even today, involve warehouse schema design and maintenance, periodically-executed ETL (extract-transform-load) scripts to copy and reformat data into its "analysis home" before being queried, and the understanding of a second/different data model in order to perform analyses on the warehouse's rendition of the application data. Like HTAP, this reduces the time to insight from days or hours to seconds.

The next two sections of the paper will drill down, respectively, on the Analytics Service's user model and internal technology. Before proceeding, however, it is worth noting several differences between Couchbase Server's NoSQL model and architecture and "traditional HTAP" in the relational world. One relates to scale: Like all of the services in Couchbase Server, the Analytics Service is designed to scale out horizontally on a shared-nothing cluster. Also like other Couchbase Server services, it can be scaled independently, as Figure 2 shows. The Analytics Service maintains a real-time shadow copy of the operational data that the enterprise wants to have available for analysis; the copy is because Analytics should be run on disjoint nodes of a Couchbase cluster in order to ensure performance isolation between the enterprises's operational and analytical workloads. The other difference relates to technology: Because of the intended scale, Couchbase Analytics is not an in-memory solution; it is designed for processing large amounts of NoSQL document data, documents whose individual value would not warrant an expensive memory-resident solution, but whose aggregated content is still invaluable for decision-making.

4. ANALYTICS USER VIEW

As described in Section 2, Couchbase Server's document data model is JSON. This means that its data objects are self-describing and that they can be nested and/or type-variant from object to object. To enable declarative querying over collections of such JSON objects, the Analytics Service provides N1QL, specifically N1QL for Analytics[‡] [20]. For the remainder of the paper we will just use

[†]For planned analytical queries, e.g., daily reports, another option in Couchbase Server is to create a covering index for such queries. This amounts to a materialized view approach.

[‡]There are minor differences between N1QL for Query, which the Query Service supports, and N1QL for Analytics. These differences have been shrinking over time and will continue to do so.

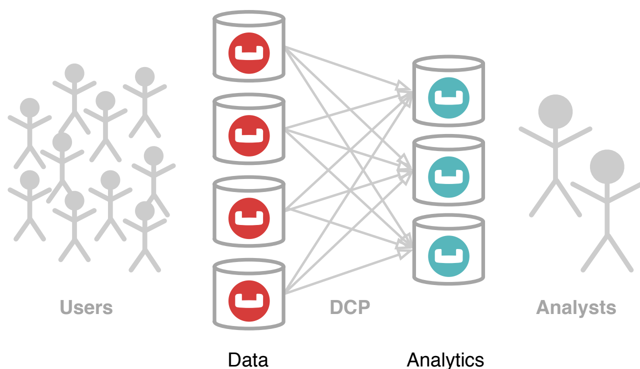


Figure 2: Bringing hybrid workloads to NoSQL

N1QL to refer to the query language. N1QL is based on SQL++ [10], a generalization of SQL for non-1NF and potentially schema-less data that was designed by researchers at UC San Diego [22]. In this section of the paper we will look in more detail at the impact of JSON as a data model and the way that its semistructured nature is addressed by N1QL. We will also see how little effort it takes to make data available through the Analytics Service in a Couchbase cluster.

4.1 Getting data flowing (no ETL)

```
CREATE DATASET customers ON salesbucket
               WHERE `type` = "customer";
CREATE DATASET orders ON salesbucket
               WHERE `type` = "order";
CONNECT LINK Local;
```

Query 1: Setup DDL (No ETL!)

The Couchbase Data Service, as mentioned earlier, currently provides the notion of a *bucket* as the container for storing and accessing documents. The Analytics Service offers more fine-grained control; its universe of data is divided into *dataverses*, and each dataverse can contain any number of *datasets* for managing its documents. A default dataverse is provided for applications that do not feel a need to have more than one namespace to organize their data.

Query 1 shows N1QL DDL statements that arrange for the customer and order documents residing in a Data Service bucket named `salesbucket` to be available, respectively, in two Analytics datasets named `customers` and `orders` in the default dataverse. To make the world friendlier for analysts, these statements choose to route documents that have a `type` field containing the value `"customer"` to the dataset named `customers`, similarly routing the `"order"` typed objects to the `orders` dataset. (Such routing is optional; one can simply route all data in a Data Service bucket to a single dataset in the Analytics Service if desired.) The `CONNECT LINK` statement, once issued, initiates the shadowing of data – the relevant Data Service data will then flow into the Analytics Service, as will all subsequent changes to that data. No ETL is required – and in the steady state, the data content in these two Analytics Service shadow datasets will track the corresponding Data Service data in real time.

4.2 N1QL

4.2.1 *Data model: NoSQL vs. SQL*

Data Examples 1 and 2 show some examples of what typical JSON data looks like and thus how it differs from SQL data. The customer objects in Data Example 1 contain a nested object `address` and the fields of this object are different in the two customer instances shown; the foreign customer's address has no `zipcode` field. The order objects in Data Example 2 contain a nested array of objects `items` so orders are decidedly non-1NF in nature. They also have field differences; one of the objects has a `ship_date` field, while that field is missing in the other object. In a normalized or "flat" relational schema, items would require their own table, linked by value back to their containing orders, and any field differences between customer or order instances would require pre-declaration of all possible fields with `NULL ALLOWED` annotations. If it were decided that an application needed to add additional fields, this would require a schema change in the relational world as well as a change in any downstream ETL scripts. With this background, we can now look in more depth at the query side of N1QL.

```
{
  "custid":"C37",
  "name":"T. Hanks",
  "address":{
    "street":"120 Harbor Blvd.",
    "city":"Boston, MA",
    "zipcode":"02115"
  },
  "rating":750
}
  ...
{
  "custid":"C47",
  "name":"S. Lauren",
  "address":{
    "street":"17 Rue d'Antibes",
    "city":"Cannes, France"
  },
  "rating":625
}
```

Data Example 1: Customers

```
{
  "orderno":1004,
  "custid":"C35",
  "order_date":"2017-07-10",
  "ship_date":"2017-07-15",
  "items":[
    {
      "itemno":680,
      "qty":6,
      "price":9.99
    },
    {
      "itemno":195,
      "qty":4,
      "price":35.00
    }
  ]
}
  ...
{
  "orderno":1008,
  "custid":"C13",
  "order_date":"2017-10-13",
  "items":[
    {
      "itemno":460,
      "qty":20,
      "price":99.99
    }
  ]
}
```

Data Example 2: Orders

4.2.2 *SQL-based query basics*

N1QL, with its basis in SQL++, is a thoughtful, semistructured generalization of standard relational SQL. As such, its design seeks

to match SQL to the extent possible for “flat”, regular data, differing only where necessary due to the lack of a predefined schema or due to JSON’s more general data model and its support for nested objects and collection-valued fields. Query 2 shows an example of a standard SQL query that is also a N1QL query. This query counts the orders from customers with a rating of over 500 by `order_date`, printing the results for those dates having at least one order — and its N1QL and SQL versions are identical. This makes it relatively easy for an analyst familiar with SQL-based data analysis to conduct the same sorts of analyses on their Couchbase NoSQL data.

```
SELECT o.order_date, count(*) AS cnt
FROM orders o, customers c
WHERE o.custid = c.custid
  AND c.rating > 500
GROUP BY o.order_date
HAVING count(*) > 0
ORDER BY o.order_date DESC
LIMIT 3;
```

Query 2: Standard SQL Query

4.2.3 *Missing information*

One area where the relational world and the JSON world begin to diverge is in regards to their handling of missing information. As mentioned earlier, SQL requires all of the fields of all of its tables to be declared. If information can be missing, this possibility must be anticipated, and the relevant fields must be declared as `NULL ALLOWED` fields. JSON is both more forgiving and a bit more complicated – depending on the desire of an application developer, a field may be either null-valued (`NULL`, as in SQL) or altogether absent (`MISSING`). Because both are possible in JSON, both must be supported and discernable in N1QL.

To address JSON’s more diverse notion of missing information, N1QL supports a four-valued logic [10] rather than the three-valued logic of SQL, and it includes `IS MISSING` as well as `IS NULL` among its missing-value-testing predicates. It also supports `IS UNKNOWN`, which covers both possibilities at once. The `WHERE` clause in Query 3 illustrates how this might be put to good use for our example data; it prints the order number, customer id, order date, and number of ordered items for those orders whose `ship_date` is missing. (It also illustrates another beyond-SQL feature of N1QL, namely the availability of functions to compute simple aggregates on set-valued arguments, such as the `array_count` function in this example.)

```
SELECT o.orderno, o.custid, o.order_date,
       array_count(o.items) AS size
FROM orders o
WHERE o.ship_date IS MISSING
```

Query 3: Missing Information

4.2.4 *Nesting data...*

A major difference between the worlds of SQL and JSON is the nesting of data allowed by JSON. N1QL thus supports the construction of JSON objects that are nested in nature. Query 4 illustrates one of the ways in which this can be done. This query identifies customers with an address that has a `zipcode` field with value `"02116"` (note the path expression to traverse nested objects) – and for each one, it forms a new object with a field `CustomerName` that has their name and a second, array-valued field `Orders` that contains their order numbers, populated via a subquery. The `VALUE` keyword in the outermost query requests that the `SELECT` results be JSON values (vs. objects), but the query then illustrates the use of an explicit JSON object constructor to form the desired object for each qualifying customer. (Saying `SELECT` and omitting `VALUE` and the curly-bracketed constructor syntax would have led to an equivalent query result.) The `VALUE` keyword in the inner subquery requests that the results that it returns for each customer to be an array of the primitive `orderno` values matching the customer rather than an array of objects with `"orderno"` fields containing those values. In SQL, the query results are always records; with JSON, however, SQL++ queries can return primitives, arrays (or multisets[§]), or objects [22].

```
SELECT VALUE {
  "CustomerName":c.name,
  "Orders":(SELECT VALUE o.orderno FROM orders AS o
            WHERE o.custid = c.custid)
}
FROM customers AS c
WHERE c.address.zipcode = "02115";
```

Query 4: Forming Nested Data

4.2.5 *... and unnesting data*

Like “what goes up, must come down” in the physical world, the NoSQL world has “what can be nested must be unnested.” To analyze and manipulate nested data, such as orders with their contained items, N1QL supports the `UNNEST`ing of nested data. This is illustrated in the next query, Query 5. This query produces a list of objects with fields `orderno`, `order_date`, `item_number`, and `quantity`, with an object in its result for each item and for each order where the order item has a quantity greater than 100 in the containing order. The unnesting of items is expressed via the `FROM` clause in the query. Saying “... `FROM orders AS o, o.items AS i` ...” leads to one binding of the query variables `o` and `i` for each order and its `WHERE`-qualifying items. The `FROM` clause items in a N1QL query are correlated left to right, so `o.items` refers to the items in each qualifying order `o`. An equivalent `FROM` clause based on a more explicit syntax, akin to SQL’s explicit JOIN syntax, is also supported, and in this example it would read: “`... FROM orders AS o UNNEST o.items AS i ...`”.

```
SELECT o.orderno,
       o.order_date,
       i.itemno AS item_number,
       i.qty AS quantity
FROM orders AS o, o.items AS i
WHERE i.qty > 100
ORDER BY o.orderno, item_number;
```

Query 5: Unnesting Nested Data

4.2.6 *Grouping and aggregation*

For data analysis, grouping and aggregation are key pieces of the query language puzzle. In SQL, groups of tuples actually lie outside the (flat) relational data model, making it difficult to explain the `GROUP BY` construct and its restrictions and semantics to newcomers to the language. In contrast, in the NoSQL world, nested JSON data and nested collections in particular are supported and very natural. As a result, grouping and aggregation are separate concepts and thus separate capabilities in N1QL. This aspect of the

[§] A multiset is an unordered collection, whereas an array is ordered. Internally, the SQL++ data model is at work, supporting both, only retaining order when the input is an ordered array or the query involves the use of ORDER BY. When serializing results in JSON form, however, multisets are converted to arrays since JSON lacks the unordered collection concept.

language is treated in some depth in [10], and it is most easily illustrated here via an example. Query 6 and its corresponding query result in Data Example 3 show some of the language's power in this regard. For each matching customer/order pair, this query groups the pairs by customer city – but instead of just aggregating (e.g., counting orders by city) as would be necessary in SQL, this query uses a `GROUP AS` clause to return the groups themselves. It names the group field `g`, returning result objects with two fields, `city` and `g`, for each city. Each group's value is a collection of objects with a pair of fields – `c` and `o` – that are named after the corresponding `FROM`-clause variables in the query.

In N1QL, thanks to its ability to handle nested data, the aggregates themselves are just regular functions – as opposed to requiring the use of a difficult-to-explain magic SQL syntax and a special notion of "aggregate functions". An example is the `array_count` function used in Query 3. N1QL has one `array_xxx` function for each of the SQL aggregates, and as arguments each one takes an array (or multiset) of values and performs the indicated aggregation, as illustrated in that earlier example. (The SQL syntax shown in Query 2 is supported for convenience and familiarity; internally it is rewritten by the N1QL engine into its more explicit native NoSQL form.)

```
SELECT c.address.city, g
FROM customers AS c, orders AS o
WHERE c.custid = o.custid
GROUP BY c.address.city GROUP AS g;
```

Query 6: Grouping and Nesting

```
[
 {
   "city": "Boston, MA",
   "g": [ {
       "c": {
         "address": { "city": "Boston, MA",     },
         "custid": "C35", "name": "J. Roberts",
         "rating": 565
       },
       "o": {
         "custid": "C35",
         "items": [
           { "itemno": 680, "price": 9.99, "qty": 6 },
           { "itemno": 195, "price": 35, "qty": 4 } ],
         "order_date": "2017-07-10", "orderno": 1004,
         "ship_date": "2017-07-15"
       }
     },
     {
       "c": {
         "address": { "city": "Boston, MA",     },
         "custid": "C37", "name": "T. Hanks",
         "rating": 750
       },
       "o": {
         "custid": "C37",
         "items": [
           { "itemno": 460, "price": 99.98, "qty": 2 },
           { "itemno": 347, "price": 22, "qty": 120 },
           { "itemno": 780, "price": 1500, "qty": 1 },
           { "itemno": 375, "price": 149.98, "qty": 2 }
         ],
         "order_date": "2017-08-30", "orderno": 1005
       }
     }
   ]
 },
 . . .
]
```

Data Example 3: Grouping Query Result

4.2.7 WINDOWed analytics

Analytical window functions are an advanced feature of the SQL language that will be supported in the next release of Couchbase Analytics. As in SQL, the `OVER` clause associated with a function call defines how the input data is partitioned, ordered, and how window frames are computed. The specified function is then evaluated over the input dataset and produces one output value for each input item in each partition. Unlike SQL, in N1QL, window function calls can appear not only in `SELECT` and `ORDER BY` clauses, but also in other clauses such as `WHERE`, `HAVING` and `LET` clauses. This is possible due to SQL++ semantics that specify that each clause consumes and produces a bag of binding tuples. As a result, a window function is simply computed on the binding tuples that are input to the clause containing the function call irrespective of the clause's kind. SQL aggregate functions that appear in window function calls are rewritten into their explicit `array_*` forms in a manner similar to SQL++'s `GROUP BY` processing. An example of window function use appears in Query 7. This query lists the names of customers that are in the top quarter, by rating, in their zip codes. The quarter number is computed by calling the `ntile` window function, which receives data partitioned by the `zipcode` field and ordered by descending `rating` values within those partitions.

```
SELECT c.name
FROM customers AS c
WHERE ntile(4) OVER (PARTITION BY c.address.zipcode
  ORDER BY c.rating DESC) = 1;
```

Query 7: Window Query

5. ANALYTICS UNDER THE HOOD

The design goal for the Couchbase Analytics Service is to support a (modest) number of data analysts concurrently posing complex N1QL queries against the operational data model in order to gain fast insights into key aspects of their business. In addition, the Analytics Service design aims to provide linear scale-up and speedup so that analysts can linearly boost the data handling capability of the service and/or its query performance by incrementally adding more nodes to the Analytics portion of their Couchbase Server cluster. In this section of the paper we take a look under the hood to see how the architecture of Couchbase Analytics addresses the resulting requirements.

5.1 Architecture

Figure 3 depicts the shared-nothing architecture of the Couchbase Analytics Service. Working bottom up, each node in the Analytics portion of a Couchbase cluster has its own locally-attached storage, and each of its datasets is hash-partitioned (sharded) across the cluster by key. The underlying storage technology used for Analytics datasets and indexes is based on LSM (Log-structured merge) B+ trees [21].

Moving up a level, each node runs an instance of Hyracks [7], a dataflow query engine designed to coordinate the execution of parallel queries. Each node also has a DCP listener that listens for operational data mutations from the Data Service and reflects them in Analytics' shadow datasets. The Data Service is currently the source of all Analytics-resident data, as the Analytics Service contains only those datasets and indexes needed to support the performance-isolated parallel query processing that enables it to deliver on its complex query promises.

Moving up another level, each node also runs an instance of the Analytics Service's HTTP-based client interface, accepting incoming requests and directing them to one of the nodes in the Analytics

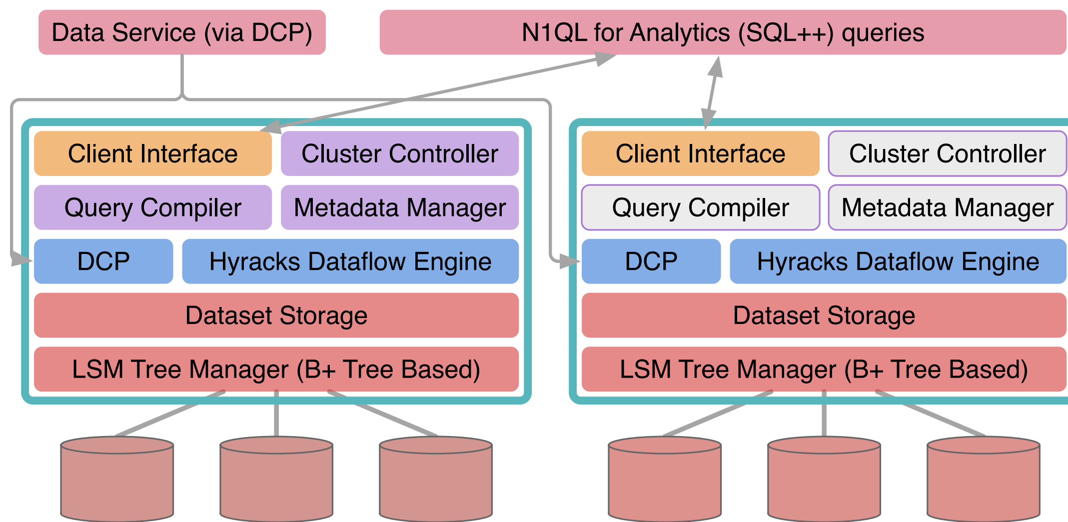


Figure 3: High-Level Architecture

subcluster that is designated as the current cluster controller. Its responsibilities include keeping watch on the overall state of the running and waiting queries, implementing admission control, and monitoring their resource consumption and progress. Another responsibility is to run an instance of the N1QL query compiler. Any node can potentially take on this role, as indicated by the (light gray) presence of the components fulfilling these reponsibilities on other nodes in the figure. In addition, one node is designated as the Metadata Manager and maintains the Analytics Service's system catalogs (which have backup replicas on all nodes in order to prevent metadata loss in the event of a failure). All of the service's dataverses, datasets, indexes, data types, functions, and other artifacts are catalogued in its metadata datasets.

Moving to the top of Figure 3, we find two main connection points to the Analytics Service. One connection is from the Data Service via DCP, as discussed above. The other is the HTTP-based query API itself, the API through which new analytical queries arrive and their results are returned to the requesting clients.

5.2 Data Ingestion and Storage

In this section we describe where Analytics Service data comes from and how it is stored and managed on arrival.

5.2.1 *Incoming! (DCP)*

As mentioned earlier, any mutation that happens to an object in the Data Service must be propagated to all other services in the system that need to know. Couchbase Server's DCP protocol keeps all of the different components in sync and can move data between the components at high speed. DCP lies at the heart of Couchbase Server and supports its memory-first architecture by decoupling potential I/O bottlenecks from many critical functions. Each node running the Analytics Service includes a DCP listener that monitors the DCP stream for relevant changes and keeps its shadow datasets up to date. As Figure 2 indicated, this is a parallel activity, with all Analytics nodes potentially receiving DCP events from any Data Service node since they reside on different nodes and are independently sharded across their respective Couchbase Server subclusters.

Since the Analytics Service shadows the Data Service for content, DCP is also the fallback in the event of a failure that impacts a dataset. When such a failure occurs, the Analytics Service is able to use DCP to re-request the required data from the Data Service. DCP can also help applications to ensure that the data that they read is sufficiently current, as a client can optionally request that its query proceed to execution only when the content of its target datasets are caught up to the point on the Data Service DCP timeline when their query was submitted.

5.2.2 *Storage and Indexing*

The top of figure 4 shows how data is distributed across the Analytics Service's subcluster. An Analytics node can have one or more storage partitions, and by default, each storage partition holds one logical (hashed) partition of each dataset. Zooming in further, each dataset consists of a primary index whose structure is based on an LSM B+ tree [21, 19] variant that stores the dataset's documents indexed by key. A dataset's LSM primary index is made up of multiple components, including an in-memory component (the leftmost component in the figure) – where new objects (or delete operations) are first stored and then later flushed as a new disk component – and a time-ordered series of disk components. Each disk component is immutable and has an associated Bloom filter to help prevent unnecessary component searches for a given key. Non-point queries must search all components, so disk components are periodically merged according to a merge policy to form fewer larger components to make such queries less expensive [1].

In addition to the primary index, a dataset can have zero or more associated secondary indexes, as Figure 4 indicates. The Analytics Service uses a local indexing strategy, so each secondary index is co-partitioned with its primary index – all secondary index entries refer (by key) to locally stored primary objects. In general, a secondary index can be built on any field(s) of a dataset's objects. Also available, as indicated in the figure, is the option to create a primary key index, essentially just an indexed list of the keys in the dataset, to accelerate certain operations (such as `COUNT(*)` queries and key presence checks). Transaction support in the Analytics Service is currently object-level; thanks to the use of local indexing, secondary index entries are kept transactionally consistent with the referenced objects in the primary index.

5.2.3 *Compression*

When dealing with very large volumes of data, complex queries can be I/O-bound depending on what they do and on their corre-

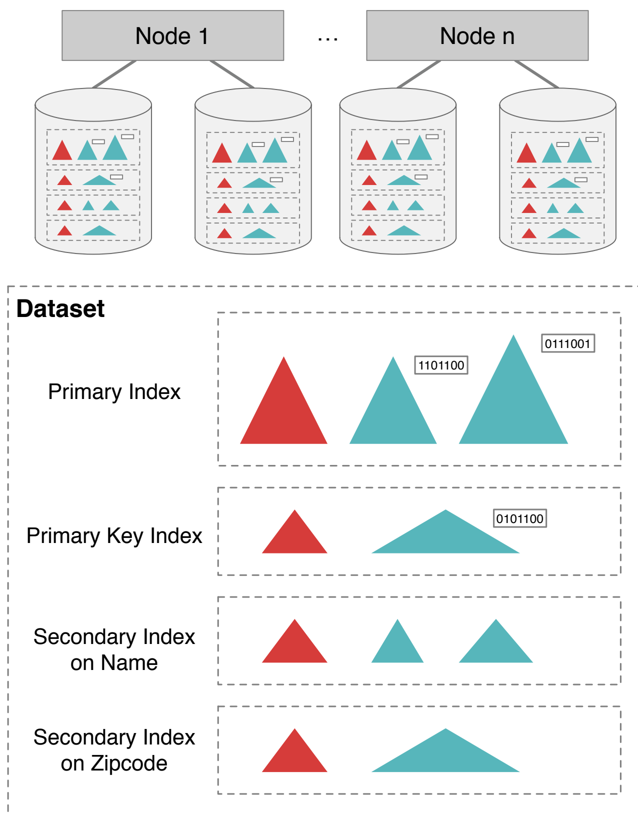


Figure 4: Datasets and Indexes

sponding CPU-intensity. In the I/O-bound case, reducing the on-disk footprint of a dataset's primary index can significantly improve query performance. To this end, the next release of the Analytics Service will include support for data compression. To minimize the code impact of compression on the rest of the system, compression occurs at the page level, and data in the system's buffer cache is compressed as it is being flushed or merged but is uncompressed when it is read into the buffer cache and thus when it is being operated on by the various physical query operators.

5.3 Local Big Data operators

From the very beginning, the Analytics query engine has aimed to support "Big Data" – data whose per-node volume exceeds, possibly by a large margin, the cluster's per-node memory resources. All of the engine's operators for performing selections, joins, sorting, grouping, and windowing are designed to work strictly within a compiler-allotted memory budget and to spill portions of data dynamically to and from disk if/as needed during their evaluation on large operands. More details on exactly how this works for each memory-intensive operator can be found in [18]; we provide a brief summary here for the key operators.

5.3.1 Selections

Queries involving selections are sent to all nodes for evaluation on their partitions of the target dataset. Local evaluation can involve either a full primary dataset scan or the use of one or more indexes. Secondary indexes map secondary keys (e.g., `customer.rating`) to primary keys; when a secondary index is utilized, it is searched and then the resulting primary keys are sorted before being used to locally retrieve the target objects in order to reduce the number of I/Os needed for their retrieval. If several indexes match a query, the optimizer will engage them all and intersect the resulting primary key lists to identify candidates that satisfy all of the indexed conditions. (These behaviors can also be influenced with hints in the event that an especially knowledgeable user wants to discourage the use of a particular index for their query.)

5.3.2 Joins

The Analytics Service currently supports three of the classic local join methods: a dynamic variant of hybrid hash join [23], index nested-loop join, and block nested-loop join. The dynamic hash join algorithm is the algorithm of choice by default for equality-based joins, as it has good/stable cost characteristics when joining large volumes of data (as is expected to occur in many Analytics use cases). In the event that its "build"-side input data fits in memory, it behaves like an in-memory hash join, and if not, it spills some of the build input and corresponding "probe"-side objects to disk for later joining [18]. For non-equality joins, the join method chosen will be block nested-loop. Both dynamic hash join and block nested-loop join operate strictly within their given operator memory budgets; they will use as much of the given memory as possible to execute the join efficiently, but will not exceed their given allocation. Also available, and accessible via a query hint, is an index nested-loop method (which is not memory-intensive) for use when one side of a given equi-join is known to be small and the other side is indexed on one or more of the join field(s).

5.3.3 External sorting

Local sort operations in Couchbase Analytics are performed using a mostly traditional external sort algorithm in which input is incrementally read, the allocated memory budget is filled with incoming objects, and the memory-load of objects (actually pointers to them) are sorted in memory and streamed to form a sorted runfile [18]. If the input is sufficiently small, an in-memory sort results; otherwise this run formation process is repeated until the input stream is exhausted, and is then followed by a merge phase where a multi-way merge of sorted runs (using as much of the memory budget as needed) is performed (possibly recursively, for very large data) to compute the final sorted output. In the event that a query includes a LIMIT clause, the N1QL optimizer will attempt to push the LIMIT operation into the sort (which yields tremendous savings for top-K queries when K is small).

5.3.4 Grouped aggregation

Couchbase Analytics supports two algorithms for grouped aggregation, hash-based and pre-sorted. To evaluate a grouped aggregate, Analytics by default will choose a generally robust pre-sorted strategy wherein the input data is first sorted on the grouping field(s) and then the aggregation can proceed in constant memory, outputting each group result when it encounters the first object outside the currently active group. Also available, via a hint, is a hash-based strategy that works particularly well for low-cardinality grouping fields. The latter strategy is roughly similar to dynamic hash join in its behavior with respect to memory use and handling of large inputs; more details can again be found in [18].

5.3.5 Presorted windowing

Local window function evaluation in the Analytics query engine is performed by a window operator that is able to evaluate one or more window functions at a time. The query compiler determines which functions can be evaluated by the same window operator and which would require a separate operator based on their window specifications. Each window operator expects to receive its input

clustered according to the `PARTITION BY` sub-clause, with each partition sorted as specified by the `ORDER BY` sub-clause of the `OVER` clause. These requirements are satisfied by the query compiler, as it inserts sort operations whenever necessary. A window operator's execution proceeds by identifying window partitions and computing a window frame for each item in each partition; it then evaluates a window function on the frame. Some window functions, such as `row_number()`, do not operate on window frames and are instead evaluated as running aggregates on every item in each partition. As with the other memory-intensive operators, the window operator works within the memory budget given to it by the query compiler, so if a window function or a frame specification requires partition materialization, the partition data is kept in memory but spilled to disk if necessary.

5.4 Parallel query processing

To handle complex analytics queries efficiently, and to deliver the desired scale-up and speed-up properties, the Analytics Service employs the same kinds of state-of-the-art, shared-nothing MPP (massively parallel processing) based query processing strategies [14] found in traditional parallel relational database products (e.g., Teradata, DB2 Parallel Edition, or Microsoft PDW). The Analytics query processing stack has adapted these time-tested partitioned-parallel query execution strategies to operate over NoSQL data. In what follows we will look in more detail at some of the relevant strategies.

5.4.1 Query optimization

To process a given request from an analyst, the Analytics Service takes in their N1QL query and converts it into an expression tree that is then handed to the system's SQL++ query optimizer. This optimizer is based on rule-based optimization technology described in [6], wherein each query is turned into an Algebricks algebraic program – including traditional relational algebra-like operators as well as extensions to deal with semistructured and nested data – that is then optimized via algebraic rewrite rules that reorder the Algebricks operators and introduce partitioned parallelism for scalable execution. On the parallelism front, the optimizer was strongly influenced by the recent work reported in [9], which introduced a formal framework for keeping track of how data is partitioned, grouped, and ordered, both locally and globally; this helps the optimizer to not move (repartition) data that is already appropriately partitioned for a given operation. After logical and physical optimization, a code generation phase translates the resulting physical query plan into a corresponding Hyracks job that is handed to Hyracks to compute the desired query result.

While the detailed workings of the SQL++ query optimizer are outside the scope of this paper, some of its attributes are worthy of calling out here. The Algebricks layer receives a translated logical SQL++ query plan from the upper layer and performs rule-based optimizations. A given rule can be assigned to multiple rule sets; based on the configuration of a rule set, each rule set can be applied repeatedly until no rule in the set can further transform the plan. For logical plan transformation, there are currently 10 rule sets and 111 rules (including multiple assignments of a rule to different rule sets). After performing logical optimization, the physical optimization phase of Algebricks selects the physical operators for each logical operator in the plan. For example, for a logical join operator, a dynamic hash join or block nested-loop join can be chosen based on the join predicate. There are 3 rule sets and 38 rules involved in the physical optimization phase. After the physical optimization process is done, the Algebricks layer generates the job to be executed by the Hyracks layer.

5.4.2 Parallel selection

To process a single-dataset selection query, the Analytics service utilizes the power of its sub-cluster in an "embarrassingly parallel" manner. Indexes are local so, as discussed earlier, each partition is able to compute its contribution to the result in parallel on its subset of the relevant data using the local selection algorithm described in the preceding section. When these parallel searches (or dataset partition scans, in the absence of any applicable secondary indexes) complete on all partitions, the overall query result can be returned to the user by concatenating the results from each partition.

5.4.3 Parallel grouping

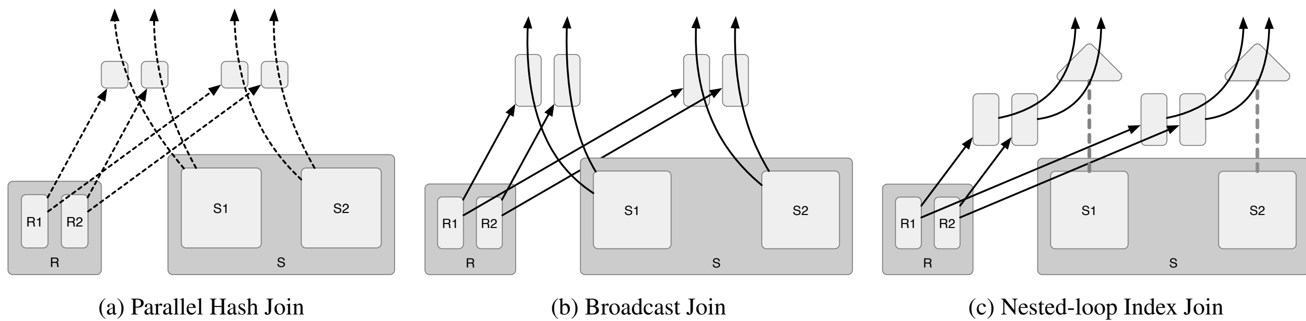
The Analytics Service employs a partitioned-parallel processing strategy to evaluate grouped aggregation queries. For such a query, e.g., to count the number of customers by city, the overall result is computed via a local step followed by a global step [16]. For the local step, each partition is tasked with computing the answer to the aggregation problem over its partition's subset of the data using one of the local algorithms described in the previous section of the paper. In our example, each partition then knows the count of its subset of customers grouped by city. These partial results are then rehashed across the cluster based on the grouping field(s), so each partition involved in processing the query receives *all* of the partial results for the groups that hash to it. Each partition can then perform another local aggregation on these incoming partial results to compute the final result per group. In our example, this would involve summing the incoming customer counts for each city-based group. After this step, the overall query result can be returned to the user by concatenating the results from each partition.

5.4.4 Parallel sorting

The Analytics Service employs a partitioned-parallel processing strategy for sorting data as well. In the current release of the system, each partition is tasked with sorting its subset of the data and passing it on to a designated single partition whose job is to merge the results of these parallel sub-sorts. The next release of the system will include an improved parallel sorting strategy in which the data being sorted is first re-partitioned based on contiguous ranges of the sorting key(s); each partition then sorts its assigned subset of the data, and the overall query result is the ordered concatenation of the results from each partition. The ranges are determined in a preliminary phase of the algorithm wherein each participant first saves its input while sampling the keys to compute a histogram of their distribution, and the resulting histograms are combined to determine the desired ranges for partitioning the data [2]. The saved inputs are then range partitioned and sorted as just described.

5.4.5 Parallel joins

The Analytics Service uses the classic partitioned-parallel hash join strategy to parallelize the execution of an equi-join – depicted in Figure 5(a). Assuming the join's input data is not partitioned in a useful way, the algorithm redistributes the data by hashing both inputs on the join key(s) – thereby ensuring that objects that should be joined will be routed to the same partition for processing – and then effects the join using the dynamic hash join described in the previous section. In slightly more detail, the "build" side of the join is first re-partitioned and fed over the network into the build step of a local hash join; each partition will then have some portion (perhaps all) of the to-be-joined build input data in memory, with the rest (if any) in overflow partitions on disk. The "probe" side of the join is then re-partitioned similarly, thus creating a pipelined parallel orchestration of a dynamic hash join. In the event that one of the inputs is already partitioned on the join key(s), e.g., because



(a) Parallel Hash Join (b) Broadcast Join (c) Nested-loop Index Join

Figure 5: Parallel Join Strategies

the join is a key/foreign key join, re-partitioning is skipped (unnecessary) for that input and communication is saved.

While this parallel hash join strategy is robust, the Analytics Service makes two other strategies available (via optional query hints) to handle two special cases that can arise, performance-wise. One is referred to as a broadcast join (Figure 5 (b)); this strategy employs a local dynamic hash join where one of the inputs (ideally a small one) is broadcast – replicated, that is – to all partitions of the other input. The broadcast input is used as the build input to the join, and once the build phase is done the participating partitions can each probe their local (full) copy of the other input in order to effect the join. The other alternative is referred to as a nested-loop index join (Figure 5 (c)), and it is the parallelization of the local nested-loop index join described earlier. In this join method, one of the inputs is broadcast (replicated) to all of the partitions of the other input, which for this strategy must be a base dataset with an index on the join key(s); as the broadcast objects arrive at each partition they are used to immediately probe the index of the other (often called "inner") dataset.

In the case of a non-equi-join, the system uses a broadcast-based version of the local block nested-loop algorithm described earlier. In this case, one of the join inputs is broadcast and temp'ed upon receipt by the participating partitions. Each participating partition then uses the local block nested-loop algorithm in order to effect its portion of the overall nested-loop join.

In all cases, regardless of the communication pattern and local join method, the overall join result can then be returned to the user by simply concatenating the join results from each partition.

5.4.6 Parallel windowing

Window operators are also evaluated in a partition-parallel fashion by the Analytics Service. Similar to `GROUP BY`, a window operator's input data is re-partitioned into independent computation partitions as specified by the `PARTITION BY` sub-clause. Each resulting computation partition receives one or more window partitions that are then locally sorted and processed according to the local mechanism described earlier. As with other operators, the re-partitioning and local sorting is only performed if necessary (i.e., if the data isn't already appropriately partitioned and/or ordered).

5.5 Memory and workload management

As indicated in the description of its local Big Data operators, the Analytics Service is careful about its use of memory and also about its degree of concurrent commitment of both memory and CPU resources.

5.5.1 Memory categories and budgeting

There are three main regions of main memory used and managed by the Analytics Service. These include in-memory component memory for datasets, a fairly traditional buffer cache, and working memory for the runtime system's memory-intensive operators.

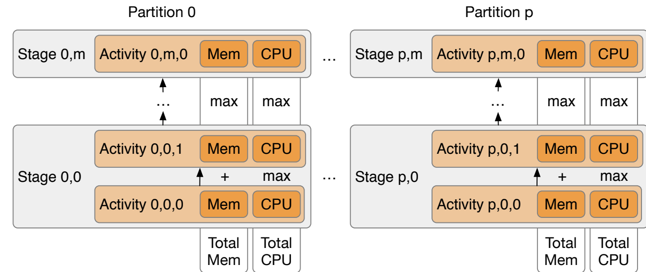


Figure 6: Resource-Based Admission Control

The in-memory component region holds the results of recent DML operations on currently active datasets. The amount of memory used for this purpose is controlled by a budget parameter that specifies the maximum amount of such memory that a dataset, including its primary index and any additional indexes, can occupy. When the in-memory component occupancy of a dataset reaches this limit, its primary index and all secondary indexes are flushed to disk and become immutable. The Analytics Service also controls the overall maximum in-memory component memory for an Analytics process. If this limit is reached and an in-memory component of a new dataset needs to be created, one of the active datasets is chosen to be flushed to disk to accommodate the new dataset.

The buffer cache is used to read disk pages from LSM disk components. Since these components are immutable, there is never a dirty page in the buffer cache that was read from a disk component. The maximum size of the buffer cache is limited by a buffer cache size parameter.

The last memory section, working memory, is used to satisfy memory allocation requests from the operators in a query execution plan. These requests require careful management since there can sometimes be a large number of operators in a complex query plan and each of the operators has different characteristics regarding its memory usage and requirements.

5.5.2 Admission control and tuning

To prevent resource over-commitment, the Analytics Service has a query admission control feature that takes two factors in each query's plan into account in order to control the number of concurrent queries. The relevant factors are the number of CPU cores (concurrent threads) requested by the query and the amount of working memory needed for its query plan. A query with aggregate working memory requirements that exceed the memory of the Analytics instance will be rejected. A query whose requirements ex-

ceed the currently available memory will be queued until one or more previous queries finish, thereby freeing up memory. For the CPU core requirement, Analytics considers the degree of parallelism induced by the number of partitions where the query's data resides. By default, queries use one execution worker thread per storage partition, but an advanced user can request a different number of worker threads (via a query parameter called parallelism), e.g., to exploit additional cores on the Analytics subcluster's nodes. The admission control policy also includes an adjustable coremultiplier parameter that can be used to control the maximum number of concurrent query workers per core (which defaults to 3).

Figure 6 provides a more detailed overview of how the admission control mechanism views its query workload. As indicated, each query's Hyracks job consists of a sequence of stages for p partitions, and each stage of a job contains anywhere from 1 to n activities. (Most stages are either single-activity or two-activity operations.) Drilling down, each activity (e.g., the build step in a dynamic hash join) has an associated operator memory requirement and CPU core requirement. The operator memory requirement of a given stage is the sum of the memory requirements of its activities over p partitions, and the CPU core requirement of a stage is the maximum of the CPU requirements of its activities for p partitions. The overall operator memory requirement of a job is then the maximum of the memory requirements of the stages for p partitions (i.e., the memory requirement at its "fattest" execution point). The CPU core requirement of a job is the maximum of the requirements of the stages for p partitions (i.e., the CPU requirement at its "widest" execution point).

5.6 Performance characteristics

As was described in Section 3, the Analytics Service complements the Query Service by making it possible to safely ask complex *ad-hoc* N1QL queries and have them answered quickly thanks to Analytics' MPP-based query processing approach. In this section we demonstrate this point empirically though a small set of experiments based on Query 8. Specifically, we will present relative performance results obtained by running this analytical query with different time window sizes to explore the operating regions and tradeoffs between the Query and Analytics services when this query arrives as an *ad-hoc* query.

For the experiments reported here, Couchbase Server was set up on a four-node cluster in which each node had an Intel Xeon e5-2680v3, 12c24t @ 2.5ghz CPU, 32GB of RAM, two PM863a SSDs, and four 1 TB 7200rpm HDDs. For the Analytics experiments, Analytics is running on all four nodes, each configured with three storage partitions on one SSD; the Data Service data nodes live elsewhere for loading purposes; the Analytics heap size is set to 20GB. For the Query service experiments, the data lives on the same four nodes as for Analytics, with three for the Indexing service and one for the Query service. In this case, node memory is split, with 16GB for data and 4GB for index. The indexes to support the Query Service were partitioned GSI indexes on the META key derived from the PK of the data.

The experiment's data was generated using the SocialGen data generator from UC Irvine [24] and the total size of the data is approximately 475 GB. From this data, Query 8 retrieves the 10 users who have sent the most messages in a social network in a specified time period. The time period is given as 2 strings in a comparable ISO8601 format, e.g. `"2018-11-28T09:57:13"` and `"2018-11-29T09:57:13"`. For the experiments here we vary the time period size from 10 seconds up to 1 month, yielding between less than 10 and 1.68M messages to be joined with their sending users.

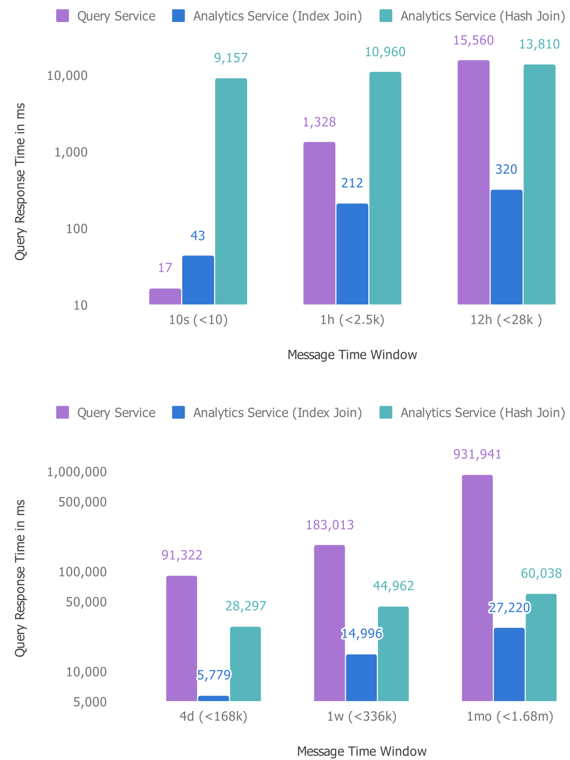


Figure 7: Performance cross-over

Query runs were assumed to be *ad-hoc*, meaning that they were run individually and that their statements were not prepared and no covering indexes or materialized views are available. For both systems, there are indexes on the keys of the datasets `users.id` and `message.author_id`, and supporting indexes on `message.author_id` (for the index join) and `message.send_time` (for the filter).

It is important to note that, in order to focus on analytical query performance, this experiment measures query latency as opposed to query throughput. The Analytics Service optimizes for latency on large data by spreading the work over multiple nodes. In contrast, the Query Service optimizes for throughput on smaller data by running many queries on a single node. Thus, just looking at query latency gives only one side of the picture. For the purpose of this particular paper, however this is the side of interest.

```
SELECT users.id, count(message) AS count
FROM gleam_users AS users
JOIN gleam_messages AS message
ON message.author_id = users.id
WHERE message.send_time BETWEEN $1 AND $2
GROUP BY users.id
ORDER BY count DESC
LIMIT 10;
```

Query 8: Join, Grouping, and Aggregation Query

The experimental results obtained for six different time windows are shown in Figure 7. (Note the use of a log scale for the query response times.) Two results are shown for Analytics, one where the join method used was a hash join, and the other where it was an index nested-loop join. The Query service used an index nested-loop join strategy. The series of results in the figure shows how the query latency varies with the size of the query. When the message time period in the query is small (10s), the Query service is the fastest

by far, outperforming the better Analytics plan by an order of magnitude and its hash join plan (which reads all of the users dataset, albeit in parallel) by several orders of magnitude. However, as the time window size increases, many more messages participate in the join. The Query Service's throughput-optimized serial query execution strategy – which was not designed or intended for this operating region – then begins losing to Analytics' parallel strategies, first to its broadcast-based index nested-loop strategy (1h window) and then to its hash-based strategy as well (crossing over at the 12h window).

As more and more messages participate in the query, the performance of Analytics' index nested-loop strategy approaches that of the hash-based strategy. With even larger time windows, it would eventually lose to it as well due to the cost of performing too many index lookups in the user dataset (as compared to scanning it in its entirety once most/all of its pages have relevant data). For this query, data, and cluster configuration, the window size for the join method crossover is about 2 years worth of messages.

6. ANALYTICS AND APACHE ASTERIXDB

The core of Couchbase Server's Analytics Service is based on the query compilation, query execution, and storage technologies of the Apache AsterixDB project [3].

6.1 The origin story

Couchbase's relationship with Apache AsterixDB began in 2015, at which time its Senior VP of Engineering was introduced by a mutual friend to members of the AsterixDB community [8]. The discussions that followed revealed a shared view of where NoSQL and Big Data platforms should be heading: a declarative, general-purpose, query-based (database) view. Fast-forward to 2016, and a decision was made to create a team within Couchbase to develop a new Analytics Service with AsterixDB as the foundation. This made sense because Couchbase's existing Data, Index, and Query services were aimed at use cases involving high volumes of operational requests, while AsterixDB's parallel database architecture seemed poised to provide the parallel query capabilities needed for complex analytics over large volumes of JSON data.

6.2 A balancing act

The famous expression "You get what you pay for" has very different meanings for Apache AsterixDB (which is free) and the Couchbase Analytics Service (an enterprise feature of Couchbase Server). The Apache AsterixDB project and code base currently serves three user bases. The first is the open source community. They expect Apache-branded Big Data software to meet certain quality requirements, particularly as it matures over time. The second is Couchbase and its paying customers, who rightfully have a high quality bar. The third are researchers and educators at universities and research institutes on multiple continents; there Apache AsterixDB provides a basis for teaching "NoSQL done right" and for empirical, systems-oriented, graduate-level database research. Many other projects lose that third role over time; we are aiming to avoid that outcome for Apache AsterixDB.

6.3 Benefits and synergy

Inviting Couchbase committers to the Apache project has been tremendously beneficial; the code base is much stronger today, student committers learn from their interactions with Couchbase's seasoned professionals, and Couchbase's uses of the system inform the students' ongoing research. Couchbase also benefits, as some of the university efforts enable the system to offer more features and improvements than it could if only Couchbase manpower was utilized. Recent examples include the forthcoming storage compression and parallel sorting improvements. It has also been beneficial for teaching; teaching use of Apache AsterixDB at the University of Washington has been a highlight, as the UW faculty and students have been a terrific source of early user feedback. Universities benefit from Apache AsterixDB's availability for use as a software laboratory; many MS and PhD thesis have resulted from the project [4].

6.4 Ongoing AsterixDB research

There are a number of AsterixDB-based projects underway that are of potential interest to Couchbase. Current UC Irvine and UC Riverside projects include work on handling expensive user-defined functions, integration of Apache AsterixDB with components of the Python data science ecosystem, resource management for complex join queries, advanced join methods for spatial or interval data, support for continuous queries, alternative semi-structured storage formats, and various optimizations for LSM storage systems.

7. CONCLUSION

This paper has provided a first technical tour of both the "externals" and internals of the Couchbase Analytics Service. This new service is the latest addition to the set of services that comprise Couchbase Server, a highly scalable document-oriented database management system that brings to the NoSQL world the same sort of declarative and scale-independent data management capabilities that relational database users have enjoyed for decades. Its Query Service supports high volumes of low-latency queries and updates for JSON documents, and the Analytics Service now provides complementary support for complex analytical queries (e.g., large ad hoc joins and aggregations) over the same operational data content.

Our tour of Couchbase Analytics has covered a number of different dimensions of the new service. Included on the tour was a description of how its dual-engine architecture provides performance-isolated and independently scalable operational and analytical query support. Also covered was how it eliminates ETL as a prerequisite to data analysis. We showed how N1QL for Analytics, based on SQL++, enables analysts to directly query their operational JSON data without having to first flatten it for warehouse consumption. We also provided a look under the hood at its MPP-based storage and query processing architectures, both of which are derived from and still developed based on Apache AsterixDB, an Apache open-source Big Data management system project with a number of Couchbase committers.

The functionality described here has opened up new possibilities for Couchbase Server usage, and more is around the corner. Planned additions that are under current development or visible on the horizon include support for user-defined functions (including both N1QL and external UDFs), the ability to use a single analytical cluster to analyze data from multiple operational clusters, and generalizations for N1QL of various additional analytical features drawn from the SQL standard.

8. ACKNOWLEDGMENTS

The authors would like to acknowledge the significant technical contributions of past Analytics team members Yingyi Bu and Abdullah Alamoudi as well as the Apache AsterixDB community who built the intial open-source system and continues to evolve it. We would also like to acknowledge Yannis Papakonstantinou for his work on the design and evolution of SQL++. Finally, we would also like to offer a special thanks to Don Chamberlin, a.k.a. the "Father of SQL", who has provided significant technical assistance and feedback related to the design of SQL++.

9. REFERENCES

[1] S. Alsubaiee, A. Behm, V. R. Borkar, Z. Heilbron, Y.-S. Kim, M. J. Carey, M. Dreseler, and C. Li. Storage management in asterixdb. *PVLDB*, 7(10):841–852, 2014.

[2] A. Alsuliman. Optimizing external parallel sorting in AsterixDB. M.S. Thesis, Department of Computer Science, University of California, Irvine, 2018.

[3] Apache AsterixDB, http://asterixdb.apache.org.

[4] ASTERIX, http://asterix.ics.uci.edu.

[5] D. Borkar, R. Mayuram, G. Sangudi, and M. J. Carey. Have your data and query it too: From key-value caching to Big Data management. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD), San Francisco, CA, USA, June 26 - July 01, 2016*, pages 239–251.

[6] V. Borkar, Y. Bu, E. P. Carman, Jr., N. Onose, T. Westmann, P. Pirzadeh, M. Carey, and V. Tsotras. Algebricks: A data model-agnostic compiler backend for Big Data languages. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC)*, pages 422–433, New York, NY, USA, 2015.

[7] V. R. Borkar, M. J. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proceedings of the 27th International Conference on Data Engineering (ICDE), April 11-16*, pages 1151–1162, Hannover, Germany, 2011.

[8] M. Carey. AsterixDB mid-flight: a case study in building systems in academia. In *Proceedings of the 35th International Conference on Data Engineering (ICDE), April 8-11, Macao, China*, pages 1–12, 2019.

[9] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[10] D. Chamberlin. *SQL++ for SQL Users: A Tutorial*. September 2018. (Available via Amazon.com.).

[11] S. Chaudhuri, U. Dayal, and V. Narasayya. An overview of business intelligence technology. *Commun. ACM*, 54(8):88–98, Aug. 2011.

[12] E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *IBM Research Report, San Jose, California*, RJ599, 1969.

[13] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[14] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[15] T. Elliott. What is hybrid transaction/analytical processing (HTAP)? https://www.zdnet.com/article/what-is-hybrid-transactionanalytical-processing-htap/, December 15, 2014.

[16] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[17] JSON. http://www.json.org/.

[18] T. Kim, A. Behm, M. Blow, V. Borkar, Y. Bu, M. J. Carey, M. Hubail, S. Jahangiri, J. Jia, C. Li, C. Luo, I. Maxon, and P. Pirzadeh. Robust and efficient memory management in Apache AsterixDB. 2019. *Submitted for publication*.

[19] C. Luo and M. J. Carey. LSM-based Storage Techniques: A survey. *CoRR*, abs/1812.07527, 2018.

[20] Couchbase N1QL for Analytics language web page, Couchbase, Inc., https://docs.couchbase.com/server/6.0/analytics/introduction.html#n1ql-for-analytics-query-language.

[21] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, 1996.

[22] K. W. Ong, Y. Papakonstantinou, and R. Vernoux. The SQL++ semi-structured data model and query language: A capabilities survey of SQL-on-Hadoop, NoSQL and NewSQL databases. *CoRR*, abs/1405.3631, 2014.

[23] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems (TODS)*, 11(3):239–264, 1986.

[24] SocialGen, https://github.com/pouriapirz/socialGen.