Cloudy With a Chance of JSON

Murtadha Al Hubail, Ali Alsuliman, Wail Alkowaileet[†], Michael Blow, Michael Carey[*], Savyasach Enukonda, Peeyush Gupta, Santosh Hegde, Kamini Jagtiani, Abhishek Jindal, Mohammad Nawazish Khan, Mehnaz Mahin[‡], Ian Maxon, M. Muralikrishna, Keshav Murthy, Daniel Nagy, Preetham Poluparthi, Ankit Prabhu, Ritik Raj, Vijay Sarathy, Shahrzad Shirazi[‡], Utsav Singh, Hussain Towaileb, Ayush Tripathi, Janhavi Tripurwar, Bo-Chun Wang, Till Westmann
Couchbase, Inc.
Santa Clara, CA, USA

ABSTRACT

Couchbase Capella is a scalable document-oriented database service in the cloud. Its existing Capella Operational service is based on a shared-nothing architecture and supports high volumes of low-latency queries and updates for JSON documents. Its new Capella Columnar cloud service complements the Operational service. The Capella Columnar service supports complex analytical queries (e.g., ad hoc joins and aggregations) over large collections of JSON documents that can originate from a variety of Couchbase and non-Couchbase data sources and formats and can either be stored and managed by the Capella Columnar service or externally stored and accessed on demand at query time. This paper describes the new Capella Columnar service, looking both over and under the hood.

PVLDB Reference Format:
Murtadha Al Hubail, Ali Alsuliman, Wail Alkowaileet, Michael Blow, Michael Carey, Savyasach Enukonda, Peeyush Gupta, Santosh Hegde, Kamini Jagtiani, Abhishek Jindal, Mohammad Nawazish Khan, Mehnaz Mahin, Ian Maxon, M. Muralikrishna, Keshav Murthy, Daniel Nagy, Preetham Poluparthi, Ankit Prabhu, Ritik Raj, Vijay Sarathy, Shahrzad Shirazi, Utsav Singh, Hussain Towaileb, Ayush Tripathi, Janhavi Tripurwar, Bo-Chun Wang, Till Westmann. Cloudy With a Chance of JSON. PVLDB, 18(12): 4938-4950, 2025.
doi:10.14778/3750601.3750618

1 INTRODUCTION

It has been more than five decades since Ted Codd changed the face of data management with the introduction of the relational data model [21]. Codd's simple tabular view of data, related by values instead of by pointers, made it possible to design declarative query languages that enable business application developers and business analysts to interact with their databases logically rather than physically – specifying what data they want, rather than how to retrieve it. The ensuing decades of research and industrial development brought numerous innovations, including SQL [20], 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. However, the world of data and applications has changed since the days of mainframes and attached terminals that were prevalent in Codd's era.

The mission-critical applications of today demand support for millions of interactions with end-users via the Web and mobile devices, as well as the ability to quickly gain insights via a wide variety of analyses of the data exhaust and other results of these interactions. In contrast, relational database systems were initially built for workloads involving thousands of users. Having been designed to provide strict consistency and data control, they tend to lack the required degrees of agility, flexibility, and scalability. As a result, to handle a wide range of demanding use cases, many organizations end up deploying multiple types of databases, resulting in a "database sprawl" that brings with it inefficiencies, including slow times to market, poor customer experiences, IT pain (e.g., due to ETL), and for analytics, slow times to insight and intelligence.

Enter Couchbase Capella, which aims to reduce the degree of database sprawl, to minimize the mismatch between an application's view of data and the persisted database state, and to do both as a data management service in the cloud. Based on a declarative SQL-like query language, namely SQL++ [16, 19, 27], its Capella Operational service supports high volumes of low-latency query and update operations over collections of JSON documents. Its new Capella Columnar service, which is the focus of this paper, complements the Operational service. It supports complex analytical queries (ad hoc joins, aggregations, etc.) over large collections of JSON documents that can originate from a variety of (Couchbase and non-Couchbase) data sources, and its data can either be stored and managed by the Capella Columnar service or stored externally in object storage and accessed on demand at query time.

The rest of the paper is organized as follows: Section 2 discusses Capella Columnar's use of JSON as its core data model. Section 3 reviews the capabilities of the service and its conceptual architecture. Section 4 describes the service's user model. Section 5 looks under the hood, offering a tour of the techniques and technologies at work behind the scenes. Section 6 concludes the paper.

2 JSON AS A DATA MODEL

The data model for Couchbase Capella is JSON [8]. JSON itself is a lightweight text-based data format with a simple specification. By design, JSON is a self-describing data format (no schema needed)

[*]Contact author. Email: mike.carey@couchbase.com
[†]Current affiliation: Saudi National Center for AI (NCAI). Work done while at Couchbase, Inc.
[‡]Current affiliation: UC Riverside. Work done while interning at Couchbase, Inc.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750618

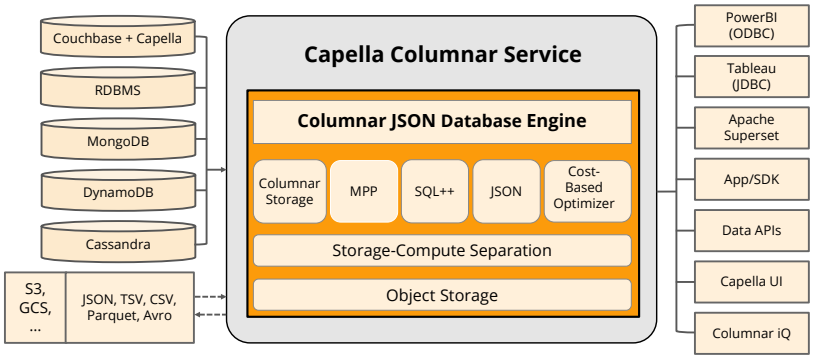

Figure 1: Overview of the Capella Columnar Service.

that is readable and writable by both humans and machines. Model-wise, JSON objects are composed of primitive types, flexible records, and arrays. JSON is in wide use for data exchange, for building and invoking web applications and web-based services, and has emerged as the defacto data format of choice for AI-infused applications.

Relational database systems are still the dominant technology in the database market [29], but today's applications are highly demanding in terms of scale, performance, schema flexibility, and online evolvability. The data that needs to be managed, in terms of its variety and regularity (or lack thereof), has led to the rise of NoSQL* database systems [18, 28, 29] and to document database systems in particular. Document databases are popular for the schema flexibility that they offer by supporting collections of semistructured objects that can be nested, heterogeneous, and/or schemaless [28]. Most document databases have adopted JSON [8] as the data model, including (in alphabetical order): Apache AsterixDB [3, 15], CosmosDB [10], Couchbase [2, 7, 14, 26], DynamoDB [1, 24], and last but certainly not least, MongoDB [9].

Given JSON's popularity, and the explosion of AI-based applications that aim to analyze and generate intelligence from every piece of available data, a vast amount of JSON data is in need of storage and analysis. JSON is a natural fit for modeling nested application objects like customers' orders or shopping carts, and it is now widely used for passing parameters and results when invoking AI services. Perhaps of equal importance, JSON is capable of modeling data coming from other, more rigid, sources. Relational tables can be easily modeled as flat collections of JSON documents [19]; likewise for data in CSV or TSV files. Data in nested schema-based formats like Parquet or Avro can also be easily modeled as JSON data. In essence, much of the world's structured and semistructured data can be neatly modeled by "putting on JSON glasses", including the data resulting from the "database sprawl" mentioned earlier.

3 CAPELLA COLUMNAR OVERVIEW

The Capella Columnar service is a near-real-time, analytically-oriented database service that has JSON as its core data model. Couchbase previously had support for performing JSON analytics [26], but it was limited to analyzing read-only shadow copies of Couchbase-managed operational data. The new Capella Columnar service has a much richer set of features, as summarized in the graphical overview of the new service in Figure 1.

The left side of Figure 1 shows the various possible data sources for the Capella Columnar service. These include remote data being streamed or bulk-copied from on-prem Couchbase Server clusters, the Capella Operational service, relational databases (Postgres and MySQL today, with SQL Server and Oracle coming soon), and NoSQL databases (MongoDB, DynamoDB, with others like Cassandra on the roadmap). Optimized access to data residing externally in cloud object stores (Amazon S3, Google Cloud Storage, and Azure Blob Storage) is also provided. From a query author's standpoint, externally resident data looks the same as internally managed data.

The middle of Figure 1 highlights the key features of Capella Columnar's database engine. The engine employs a columnar JSON storage format to enable high-performance querying and analysis of large JSON datasets. At rest, data resides in Columnar-managed object storage, and the engine's query runtime uses partitioned parallelism (also known as massively parallel processing or MPP) for its compute nodes. The architecture uses storage-compute separation for scalability. The user language provided for querying and manipulating JSON data is SQL++[16], an extension of SQL, and query planning is performed by a cost-based optimizer (CBO).

The right-hand side of Figure 1 shows the various paths through which queries from programmatic applications, interactive analysts, and other users can interact with the Capella Columnar service. These include popular BI tools for analysts, an SDK and other data APIs for applications, and the Capella Columnar query workbench UI for interactive SQL++ users, which also includes an LLM-based assistant (Columnar iQ) for developing queries using English.

Before moving on, it is worth calling out which features are new in terms of Couchbase's support for JSON analytics. SQL++, MPP execution, and support for BI tools existed in the previous Couchbase Analytics service, but were limited to operating on (row-based) copies of JSON collections managed by Couchbase Server's Data service or its Capella Operational equivalent. CBO and basic access to external data that resides in object storage were also available. Everything else in Figure 1 is new! That includes columnar storage for JSON, compute-storage separation, column-aware CBO, stream-based ingestion of data from non-Couchbase sources, and path-optimized access to accelerate queries over external data.

*NoSQL initially stood for "no SQL", but today it is commonly said to be short for "not only SQL".

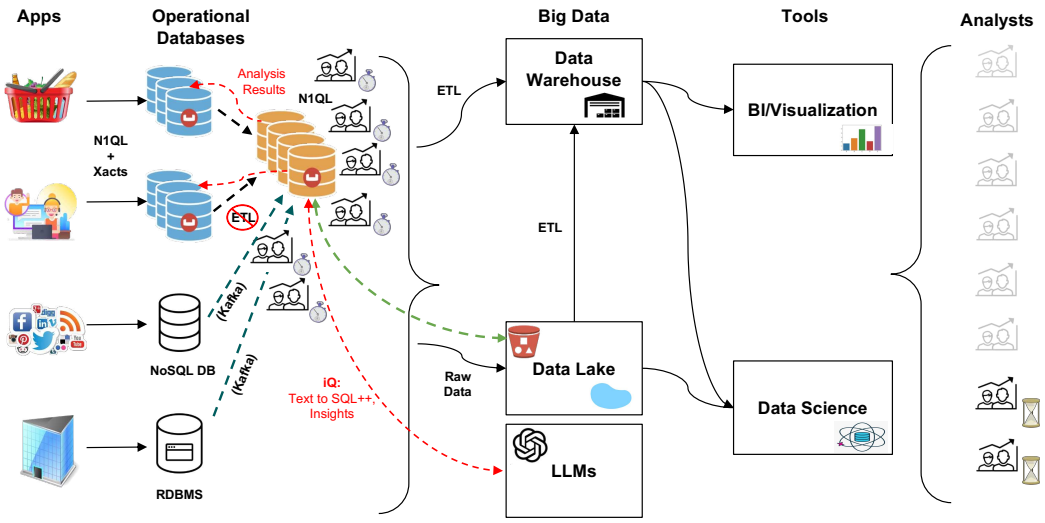

Figure 2: Columnar in the IT World.

Popping up a level, Figure 2 shows how Capella Columnar is expected to fit into the IT architecture of a typical enterprise. In its absence, analysts who need to query data coming from multiple data sources or multiple collections would typically have to wait for the relevant data to periodically find its way into a data warehouse or across a data lake. When Capella Columnar is added to the picture, data can arrive in near-real-time streams and become instantly available for analyses. Results from analyses can also be written back to a Couchbase operational database or to object storage, e.g., to support personalized web applications. Moreover, because the data model of the service is JSON, the incoming data is not required to undergo ETL-based flattening transformations – it can be analyzed "as is" in its natural form. And, due to the schema flexibility of JSON, data can fluidly evolve at its source(s) without requiring IT staff to plan and perform ALTER TABLE operations or update ETL scripts. As a result, the incoming data is immediately available for analysis and other applications. Finally, as hinted at in the figure, the iQ assistant in Columnar's query workbench can help data analysts through its support for text to SQL++ and text to graphical visualization.

4 CAPELLA COLUMNAR USER MODEL

We now describe the Capella Columnar service as seen by data analysts and other end users. Much more detail is available in [2].

4.1 Logical and Physical Resources

An organization wanting to use the Capella Columnar service for a new JSON data analytics activity will start by choosing or creating a Capella Columnar *cluster* (in the Capella cloud) with which to manage the activity's data and run its analytical queries. The cluster's size, in terms of its initial number of nodes, is specified at create time but can be scaled up or down at any time later based on the activity's workload and its performance needs. Each node brings with it a set of vCPUs (cores) as well as memory and NVRAM for caching data. In essence, a cluster is a dedicated instance of the Capella Columnar service and its underlying JSON database engine. That is, a cluster is where the data arriving from the left side of Figure 1 will live, and it is also an active entity to which query requests coming from the right side of Figure 1 will be directed.

In addition to a Capella Columnar cluster, an important global (cluster-wide) entity is a *link* to an outside data source. A link is an entity that holds the authentication credentials that Capella Columnar needs to connect to a remote or external data source. A remote link holds credentials for a streaming data source, such as a Capella Operational service instance, a remote Couchbase Server cluster, or a Kafka-based data source. An external link holds credentials needed to access data in an external object store. Creation of clusters, remote links, and external links are handled by the Couchbase Capella UI or by a REST API call. (SQL++ DDL statements exist to create links, but they are not documented and enabled for end users because link creation requests include security-related information that should not be exposed in plaintext form.)

4.2 Databases, Scopes, and Collections

The data managed by a Capella Columnar cluster is logically organized into a hierarchical namespace made up of databases, scopes, and collections. A cluster can host one or more databases, and a database, in turn, can have one or more scopes to help organize its collections, indexes, and functions. Collections are the containers for JSON objects and the targets for SQL++ queries. They can contain JSON objects of any type, with no schema being requested or required, but the collections that Columnar manages must have a declared primary key field (or fields) that is used to index and hash partition its objects. (Auto-generated UUIDs are provided as an option for collections whose objects lack a natural primary key.) As we will see by example below, Columnar provides support for several different kinds of collections.

Figure 3 shows some objects from a simple example commerce database based on extending a set of tutorial data from [19]. To illustrate the integrative power of the Capella Columnar service,

customers:

```
{ "custid": "C13",
  "name": "T. Cody",
  "address": {
     "street": "201 Main St.",
     "city": "St. Louis, MO",
     "zipcode": "63101" },
  "rating": 750
},
{ "custid": "C31",
  "name": "B. Pruitt",
  "address": {
     "street": "360 Mountain Ave.",
     "city": "St. Louis, MO",
     "zipcode": "63101" }
},
{ "custid": "C47",
  "name": "S. Logan",
  "address": {
     "street": "Via del Corso",
     "city": "Rome, Italy" },
  "rating": 625
},
...
```

orders:

```
{ "orderno": 1002,
  "custid": "C13",
  "order_date": "2020-05-01",
  "ship_date": "2020-05-03",
  "items": [
    { "itemno": 460, "qty": 95,
      "price": 29.99 },
    {"itemno": 680, "qty": 150,
      "price": 22.99 } ]
},
{ "orderno": 1003,
  "custid": "C31",
  "order_date": "2020-06-15",
  "ship_date": "2020-06-16",
  "items": [
    { "itemno": 120, "qty": 2,
      "price": 88.99 },
    { "itemno": 460, "qty": 3,
      "price": 29.99 } ]
},
{ "orderno": 1008,
  "custid": "C13",
  "order_date": "2020-10-13",
  "items": [
    { "itemno": 460,"qty": 20,
      "price": 29.99 } ]
},
...
```

products:

```
{ "itemno": 347,
  "category": ["essentials"],
  "name": "Beer Cooler Backpack",
  "manuf": "Robo Brew",
  "listprice": 25.95,
  "kind": "electric",
  "power": "D batteries"
},
{ "itemno": 375,
  "category": ["music"],
  "name": "Stratuscaster Guitar",
  "manuf": "Fender Bender",
  "listprice": 149.99
},
{ "itemno": 460,
  "category": ["music", "clothing"],
  "name": "Fender Bender T-Shirt",
  "descrip": "Extra Large T-Shirt for
Fender Bender fans",
  "manuf": "Fender Bender",
  "listprice": 34.99,
  "kind": "clothing",
  "size": "XL"
},
...
```

reviews:

```
{ "itemno": 193,
  "name": "Super Stapler",
  "rating": 5,
  "comment": "This electric stapler is the bomb",
  "custid": "C41",
  "rev_date": "2020-05-13"
},
{ "itemno": 347,
  "name": "Beer Cooler Backpack",
  "rating": 5,
  "comment": "Every camper needs one of these for
sure",
  "custid": "C41",
  "rev_date": "2020-05-13"
},
{ "itemno": 375,
  "name": "Stratuscaster Guitar",
  "rating": 4,
  "comment": "Anxiously awaiting its arrival!",
  "custid": "C37",
  "rev_date": "2020-09-07"
},
...
```

Figure 3: Simple Commerce Data Example.

let us assume this data is from Ganges.com, a hypothetical startup that wants to someday be the Amazon.com of India. Their online store is backed by a Capella Operational database. Their product inventory is stored in a corporate PostgreSQL database. Product reviews, which are lower in value but higher in volume, are stored in S3 for cost reasons. To enable analysts to integrate and utilize all of this data in near real time, Ganges.com has chosen Capella Columnar. They can start by creating a database (commerce) and several scopes (websales, inventory, and marketing) to logically group their collections.

To create the database, commerce, and its first scope, websales, they can use the following DDL commands. Here we assume that their Operational database is up and running and has a bucket (which is like a database) with a scope containing their online customers and orders collections. The AT clauses in the CREATE COLLECTION statements refer to a link to the web store's Operational cluster; we assume this link was already created using Columnar's UI or REST-based management API.

```
-- websales (from Capella Operational):
CREATE DATABASE commerce;
CREATE SCOPE commerce.websales;
CREATE COLLECTION commerce.websales.customers
   ON commerce_app_bucket.websales_scope.customers
   AT sales_app_operational_cluster_link;
CREATE COLLECTION commerce.websales.orders
   ON commerce_app_bucket.websales_scope.orders
   AT sales_app_operational_cluster_link;
```

When executed, these commands will cause Columnar collections to be created and will start flows of data which will then be continually ingested (shadowed) in real time from Ganges' online store. Analysts can now write queries involving customers and orders and they will always get current (e.g., up-to-the-minute) answers.

Similarly, the following commands create the inventory scope and products collection. Here, the data is coming from a PostgreSQL table of Ganges' products. In the previous case, the remote data was in a Couchbase system, so shadowing will be handled transparently using Couchbase's database change protocol (DCP) [14, 26]. To create shadow collections for non-Couchbase sources, such as PostgreSQL, MySQL, MongoDB, or DynamoDB, the Capella Columnar service makes use of Kafka [4]. As a result, the CREATE COLLECTION statement below includes a link to a Kafka cluster and Kafka topic that will have been created for the purpose of change data capture (CDC) from the PostgreSQL products table. It also includes details about the Kafka connector and names the field(s) to serve as the Columnar collection's primary key. (In the Couchbase-resident case, Columnar knows what the remote primary key is, so it did not need to be told in the CREATE COLLECTION statements.)

```
-- inventory (from PostgreSQL via Kafka):
CREATE SCOPE commerce.inventory;
CREATE COLLECTION commerce.inventory.products
   PRIMARY KEY (itemno: int)
   ON mysql_products_kafka_topic
   AT my_kafka_cluster_link
   WITH { "keySerializationType": "JSON",
        "valueSerializationType": "JSON",
        "cdcEnabled": "true",
        "cdcDetails": {
           "cdcSource": "POSTGRESQL",
           "cdcSourceConnector": "DEBEZIUM"
        }
      };
```

Next, the following commands can be used to create the marketing scope and the reviews collection. Here, the desired data resides in files in an S3 bucket, so the CREATE COLLECTION statement includes the S3 bucket and an S3 link. For the data itself, the path to the data files within the bucket, and also their file format†, is

†Currently supported formats include JSON, CSV/TSV, Parquet, and Avro.

specified. The resulting collection will now be queryable by an analyst just as if the data were Columnar-resident, but in this case the data will continue to live outside the service and be accessed on demand (i.e., just in time) at query time, so a query will always see the current state of the data in object storage[‡].

```
-- marketing (external in S3):
CREATE SCOPE commerce.marketing;
CREATE EXTERNAL COLLECTION commerce.marketing.reviews
   ON marketing_data_s3_bucket
   AT my_s3_link
   PATH "marketing/reviews"
   WITH { "format": "parquet" };
```

Last but not least, Capella Columnar supports standalone collections. These are normal collections whose contents are managed by the system and can be both queried and updated using SQL++. It is important to note that the preceding collections – containing data being shadowed from remote Couchbase or non-Couchbase data sources, or living externally in an object store – will be read-only (query-only) to Columnar users since they are intended to accurately reflect the current state of remote data and thus should not be able to be modified in Columnar. Standalone collections are different in this regard, as they can be updated in addition to being queried. They are intended to be the property of user(s) needing to create and manage their own data content. The following DDL command shows how a standalone collection can be created by an analyst getting ready to study a selected snapshot of the overall collection of review data:

```
-- collection for review analyses
CREATE COLLECTION commerce.marketing.myReviews
PRIMARY KEY (id: int);
```

4.3 Queries and Updates

SQL++ [16] is the query language for Capella Columnar. It has been implemented in the Apache AsterixDB project by committers from both academia (UCI, UCR) and Couchbase. Here, we walk through a set of examples to convey the nature of the language and the power it offers for data from multiple sources. A brief tutorial on SQL++ is included in [26], and a longer tutorial treatment of SQL++ for SQL developers can be found in [19].

To a first approximation, SQL++ is a superset of SQL, so it is easy for SQL-savvy data analysts to learn SQL++. Our first example is a SQL query that is also a SQL++ query. It counts the number of orders placed by customers whose rating is over 500, grouped by order date, for dates with one or more such orders.

```
-- Query 1: SQL++ is like SQL
SELECT o.order_date, count(*) AS order_cnt
FROM commerce.websales.orders o,
    commerce.websales.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;
```

The relational model is flat, whereas JSON permits nested objects, nested arrays, variant and/or missing fields, and more (see Figure 3). SQL++ extends SQL to handle JSON's richer structure. The next query illustrates some of SQL++'s extensions. First, it shows that one can write a SQL++ query starting with its FROM clause and putting its SELECT clause where it should have been in SQL, according to SQL's inventor, Don Chamberlin [19] – after all the referenced variables and fields have been defined. Second, its WHERE predicate shows how nested field values are accessed. Finally, its correlated subquery shows the use of the SELECT VALUE clause in SQL++ to return scalar values instead of JSON objects, and it shows that subqueries return arrays in SQL++ (rather than returning a single scalar value as in SQL).

```
-- Query 2: SQL++ extends SQL
USE commerce.websales;
FROM customers AS c
WHERE c.address.zipcode = "63101"
SELECT c.name,
     ( SELECT VALUE o.orderno FROM orders AS o
       WHERE o.custid = c.custid ) as orders;
```

The actual JSON result from this query will look like:

```
[ { "name": "R. Dodge",
    "orders": [1006, 1001] },
  { "name": "B. Pruitt",
    "orders": [1003] },
  { "name": "T. Cody",
    "orders": [1002, 1009, 1008, 1007 ] } ]
```

The next example query creates a list of JSON objects with fields orderno, order_date, item_number, and quantity, and it shows how SQL++ operates on nested arrays (e.g., with an existential predicate) as well as making it possible to unnest them (e.g., by including o.items in the FROM clause). An object will appear in the result for each order-item pair where (1) the order (o) in the order-item pair contains some line item with a quantity less than three and (2) the item (i) in the order-item pair itself has a quantity over 100.

```
-- Query 3: querying nested data
SELECT o.orderno, o.order_date,
      i.itemno AS item_number, i.qty AS quantity
FROM commerce.websales.orders AS o, o.items AS i
WHERE (SOME li IN o.items SATISFIES li.qty < 3)
  AND i.qty > 0
ORDER BY o.orderno, item_number;
```

Up to now our queries have been on data from Ganges' online store. The beauty of Columnar is that queries can combine data from any/all of a cluster's collections. Our last example query (for now) illustrates this point by producing an activity profile for Ganges customers residing in the U.S. (i.e., customers with a zipcode).

```
-- Query 4: profile the U.S. customers
SELECT c.custid, c.name, c.address.zipcode,
      (SELECT VALUE COUNT(*)
       FROM commerce.websales.orders o
       WHERE o.custid = c.custid)[0] AS order_cnt,
      (SELECT COUNT(*) AS num_reviews,
            AVG(r.rating) AS avg_rating
       FROM commerce.marketing.reviews r
       WHERE r.custid = c.custid)[0] AS reviews
FROM commerce.websales.customers AS c
WHERE c.address.zipcode IS NOT UNKNOWN;
```

Here is an example of an object from its results:

```
{ "custid": "C41",
  "name": "R. Dodge",
  "zipcode": "63101",
  "order_cnt": 2,
  "reviews": {"num_reviews": 3, "avg_rating": 4}
}
```

In addition to queries, of course, Columnar supports INSERT, UPSERT, DELETE, and UPDATE operations for working with standalone collections. The following example creates a scratchpad scope and a myCusts collection and INSERTs our profiling query's results into it for further analysis and/or manipulation:

[‡]Currently supported object stores include AWS S3, storage appliances that emulate S3, Google Cloud Storage (GCS), and Azure Blob Storage.

```
CREATE SCOPE commerce.scratchpad;
CREATE COLLECTION commerce.scratchpad.myCusts
    PRIMARY KEY (custid: string);
INSERT INTO commerce.scratchpad.myCusts
SELECT c.custid, c.name, c.address.zipcode,
    (SELECT VALUE COUNT(*)
     FROM commerce.websales.orders o
     WHERE o.custid = c.custid)[0] AS order_cnt,
    (SELECT COUNT(*) AS num_reviews,
          AVG(r.rating) AS avg_rating
     FROM commerce.marketing.reviews r
     WHERE r.custid = c.custid)[0] AS reviews
FROM commerce.websales.customers AS c
WHERE c.address.zipcode IS NOT UNKNOWN;
```

An attempt to repeat the same INSERT would fail since objects will exist in myCusts with conflicting primary keys. Changing INSERT to UPSERT would allow a re-run, as UPSERT overwrites existing data with incoming data when primary keys match. Note that UPSERT works by inserting or overwriting whole objects. In addition to UPSERT, Columnar supports UPDATE for making small changes to existing data without requiring entire new object values to be provided. For example, to simply update a city's name:

```
UPDATE commerce.scratchpad.myCusts
SET address.city = "Saint Louis, MO"
WHERE address.city = "St. Louis, MO";
```

DELETE operations are also supported for standalone collections. As an example, if a data analyst decided to exclude all St. Louis customers from their data analysis for some reason, they could delete them from their customer profile snapshot:

```
DELETE FROM commerce.scratchpad.myCusts
WHERE address.city = "St. Louis, MO";
```

It's worth noting that SQL++'s mutation support for Capella Columnar generalizes the familiar SQL UPDATE operation to work on collections of nested objects by allowing nested INSERT, DELETE, and UPDATE operations on nested array-valued fields.

4.4 Views and User-Defined Functions

In addition to queries and updates, Capella supports the creation of both views and user-defined functions. Its views are similar to relational views; they are defined using a SQL++ query and support all of JSON's potential for nesting and heterogeneity. Also supported are two kinds of user-defined functions (UDFs). The first kind are SQL++ UDFs, defined using a SQL++ query – they can be thought of like parameterized views and they can be optimized when queried by the query planner. The second kind of UDFs are Python UDFs, which are implemented using Python and then registered with Columnar. These external UDFs run in a fenced environment and cannot access resources outside of that environment.

As a first example here, our customer profile query could be used to define a view that will then provide an up-to-date view of Ganges' customers' activity levels whenever queried:

```
CREATE VIEW commerce.marketing.customerProfiles AS
SELECT c.custid, c.name, c.address.zipcode,
    (SELECT VALUE COUNT(*)
     FROM commerce.websales.orders o
     WHERE o.custid = c.custid)[0] AS order_cnt,
    (SELECT COUNT(*) AS num_reviews,  AVG(r.rating) AS avg_rating
     FROM commerce.marketing.reviews r
     WHERE r.custid = c.custid)[0] AS reviews
FROM commerce.websales.customers AS c
WHERE c.address.zipcode IS NOT UNKNOWN;
```

As a second example, we can create a SQL++ UDF that takes a customer id as an argument, so that calling `customerProfile("C41")` produces the customer activity profile shown earlier, as follows:

```
CREATE FUNCTION commerce.marketing.customerProfile(cid) {
  SELECT VALUE c
  FROM commerce.marketing.customerProfiles AS c
  WHERE c.custid = cid
};
```

Columnar also allows a SQL++ UDF to be labeled as a transform function (`CREATE TRANSFORM FUNCTION`) and referred to when creating a shadow collection to apply lightweight transformations (filtering and/or projection) before depositing each incoming object. This capability, think "ETL Lite", based on [25], is a novel feature of Columnar's incoming data pipelines and was requested by a number of Couchbase customers.

We now turn to Columnar's support for external functions, namely Python UDFs. Let's suppose that Ganges wants to run analytical queries on reviews and their sentiments. If they had a function `sentiment(text)` that returned a sentiment value ("positive", "neutral", or "negative") when passed a string, Ganges could run the following query to see how many reviews of each sentiment the manufacturers of the products that they sell have had:

```
SELECT p.manuf AS mfg,
     sentiment(r.comment) AS sent,
     COUNT(*) AS num
FROM commerce.inventory.products p,
   commerce.marketing.reviews r
WHERE r.itemno = p.itemno
GROUP BY p.manuf, sentiment(r.comment);
```

resulting in output like:

```
{"mfg": "Fender Bender", "sent": "negative","num": 3}
{"mfg": "Fender Bender", "sent": "neutral", "num": 2}
{"mfg": "Office Min", "sent": "positive", "num": 1}
{"mfg": "Robo Brew", "sent": "neutral","num": 1}
{"mfg": "Robo Brew", "sent": "positive", "num": 1}
```

But where can Ganges get such a function? Python UDFs to the rescue! One approach that they could take would be to hand write a function in Python, based on some knowledge of review wordings (e.g., "It's the bomb!"):

```
pos_words = ['bomb','needs']
neg_words = ['shrinks','shrunk', 'smaller']

def sentiment(arg)
   words = arg.split()
   if any(w in words for w in pos_words) and  \
     all(w not in words for w in neg_words):
    return "positive"
   if any(w in words for w in neg_words) and  \
     all(w not in words for w in pos_words):
    return "negative"
return "neutral"
```

They would then add this new Python function to their cluster's UDF library registry. Libraries are UDFs with dependencies, and they are uploaded to the cluster via a RESTful API similar to the Links API. Once uploaded, libraries become catalog artifacts, and the functions within them can be referenced in the DDL used to create a UDF. For Python this requires a module name and either a function or class and method name. To make this concrete, Ganges could define their sentiment function's SQL++ signature and make it available to be called in queries as follows:

```
CREATE FUNCTION sentiment(text)
AS "sentfuns"."sentiment" AT my_python_library;
```

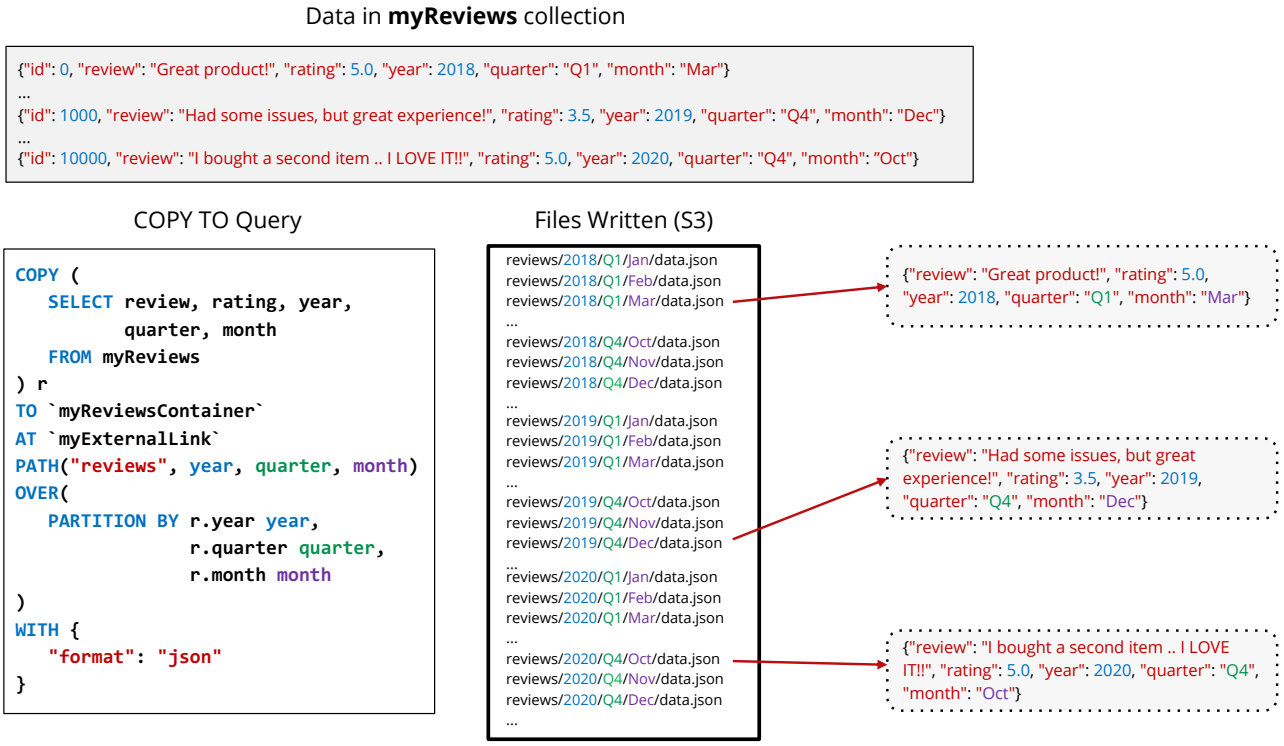

Figure 4: COPY TO for Object Storage.

While illustrative of what's involved in using the Python UDF facility for simple functions, a more realistic approach would be either to (i) use a machine learning library to train a function on a corpus of review test data and then add the trained function to the library in a similar fashion, or (ii) acquire such a Python sentiment function from an existing open source NLP library or from the emerging LLM ecosystem.

4.5 Bulk Data Import/Export

So far we have seen how data in a Columnar collection can arrive via incremental streaming or via INSERT or UPDATE statements. Bulk exporting and importing of data are also supported.

As a data export example, suppose that a Ganges analyst had created the standalone collection `myReviews` discussed at the end of Section 4.2. Figure 4 shows an example of a SQL++ COPY TO query that will export its data to an external object store. The data being exported is from the internal collection `myReviews`, and in addition to exporting the data, the novel OVER clause supported in COPY TO statements structures the results in a form that will be amenable to efficient subsetting by subsequent queries – from Columnar or from other engines with access to the data – over the resulting S3-resident external data.

For data import, Columnar has a COPY INTO statement for copying data in bulk from an external object store into a standalone collection. COPY INTO's semantics are UPSERT-based, so the copied objects will either replace existing objects with the same primary keys or be added to the collection if no existing object with the same key is present.

4.6 BI Tool Connectivity

Designed for data analysis use cases, the Capella Columnar service includes support for data analysts who would like to use one of several popular relational business intelligence (BI) tools to work with their data. BI tool support is currently available for Tableau, PowerBI, and Apache Superset. To enable "flat" relational tools to work against JSON data, Columnar provides a novel *tabular view facility* that lets power users define interrelated SQL table-like views (tabular views) on top of the database's JSON collections. These views can then be queried by regular business analysts, using their favorite BI tools, as if they were actual SQL tables.

Figure 5 provides an overview of this tabular view facility. On the right-hand side of the figure we see customers, orders, and products in JSON form, a form that a BI tool is not able to consume. On the left, however, we see a set of SQL tables that represent the same information – the tables are in 1NF, so orders and line items now appear as separate tables related by order number. The tabular view facility allows the actual collections on the right to be seen as virtual SQL tables that a BI tool can understand.

Briefly, how this works is that a CREATE VIEW statement for a tabular view differs in two ways from the regular CREATE VIEW DDL in Section 4.4: (1) Its definition specifies a flat SQL schema (i.e., column names and data types) that it should appear to have. (2) Its definition can include optional primary and foreign key information that a BI tool can use to graphically guide users who need to query and visualize joined data. A tabular view's query body is written in SQL++ and has the full power of the language to shape the view's

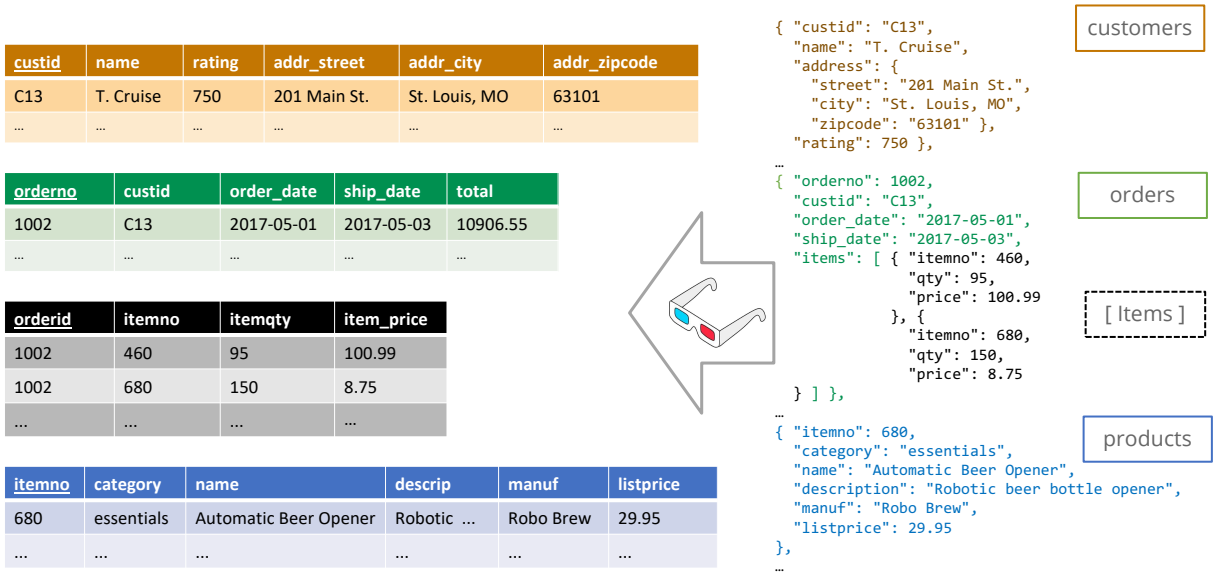

Figure 5: Tabular Views for BI Tools.

content into the desired flat form. Behind the scenes, the BI tools talk to the Columnar service via JDBC or ODBC, and the Columnar SQL++ query engine has a SQL compatibility mode that directs it to interpret queries using SQL-92 semantics rather than SQL++ semantics (e.g., when interpreting subqueries).

4.7 Columnar Query Workbench and iQ

Last but not least, in terms of what a user of the Capella Columnar service sees, is its Query Workbench and the LLM-based query and charting assistant (iQ) that it offers. Figure 6 shows a screen shot. On the left is information about the cluster's databases, scopes, and collections. On the right is the iQ panel. Above, in the middle, is the query entry panel, and below it is the panel where query results are displayed (with various options being available, e.g., JSON, tables, charts, ...). In the iQ panel on the right hand side we see that the user has selected two of the available collections (customers and orders) and has asked iQ a query in English that it responded to with a suggested SQL++ query that the user has then inspected (to verify the query's capture of their intention) and executed.

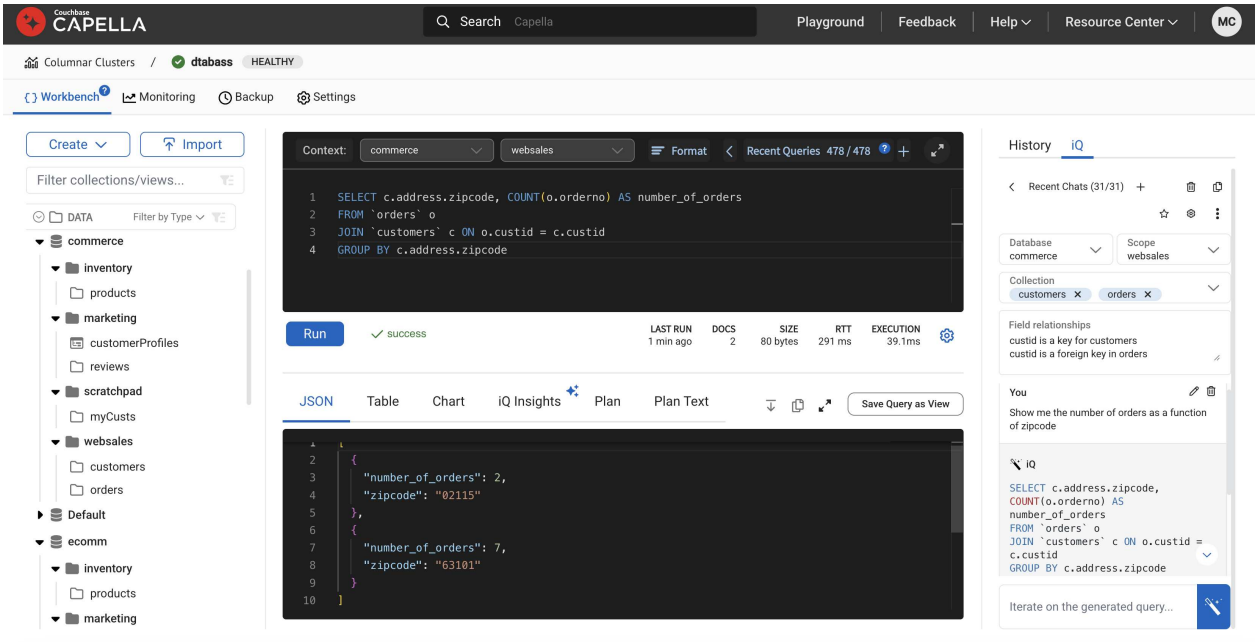

Figure 6: Columnar Workbench and iQ.

iQ bridges the gap between the user's intended business question and an executable query by converting natural language questions to SQL++. Each LLM prompt from iQ includes the SQL++ dialect preference (i.e., Capella Columnar or Capella Operational), examples of specific syntax like UDFs, and the inferred JSON schema of the user-selected collections. For each LLM prompt, iQ also includes information from the prior Q&A to create a conversational experience. If the compiler reports a syntax error for a generated SQL++ query, iQ feeds the error back to the LLM and retries, correcting most defects transparently. After execution, iQ infers the schema of the JSON result set, asks the LLM to suggest an appropriate visualization, and renders the chart automatically. It also synthesizes follow-up analytical questions and companion visualizations, enabling iterative, natural language driven data analysis and exploration. This is done sharing just the selected collections' metadata, i.e. without sharing any customer data with the LLM (openAI).

5 CAPELLA COLUMNAR UNDER THE HOOD

We now turn our attention under the hood to examine some of the key technologies that underlie the Capella Columnar service.

5.1 Columnar Storage and Indexing

Figure 7 shows how the data for a JSON collection is organized in a Capella Columnar cluster. Columnar's data organization is based on a combination of hash partitioning and compute-storage separation. The disk icons in the figure represent storage partitions, and the dotted-line associations of these icons with the compute nodes indicates which nodes are currently responsible for reading and writing which partitions' data. The objects in a collection are assigned to partitions by hashing them based on the collection's specified primary key. The data for each partition of a collection resides in a set of interrelated log-structured merge (LSM) tree indexes. The primary index contains the data objects themselves, with zero or more secondary indexes serving to map the values of an object's other attributes (secondary keys) to the primary key to speed the execution of selective queries.

Columnar's memory is divided into a buffer cache (for holding pages of on-disk components), ingestion memory (for holding pages of new in-memory components), and working memory (which provides scratch pages for operators like joins). Like the buffer cache, the ingestion memory is a shared pool of pages. When an in-memory component needs more space, it requests a page from this shared pool. When usage exceeds a certain percentage of the ingestion memory, one or more LSM flushes are scheduled to reclaim pages. Flushes are scheduled in a round-robin manner. When a flush is triggered for a collection, the in-memory components of all its indexes are flushed simultaneously; this aids in identifying a shared rollback point across the collection's indexes. Bloom filters are associated with primary index components to minimize unsuccessful primary key lookups.

Figure 8 provides a sketch of one of the most important features of the Columnar service's JSON database engine – namely, its unique binary columnar data representation, AMAX [11], for storing collections of JSON data objects.The figure shows roughly how AMAX infers and separates the schema information for a collection from the data values of its objects in an LSM-based storage world. When new data is added to the primary index in that world, it first goes into the index's in-memory LSM component. When that component fills, it will be flushed and will become the index's newest immutable disk component. During this phase of the component's lifecycle, when the data is flushed, the schema of its content will be observed and recorded once for the component. Also, at this time, the component's data content will be transformed into AMAX's generalized columnar JSON format, as indicated in caricature form at the bottom of Figure 8.

5.2 Compute-Storage Separation and Scaling

Figure 9 provides a high-level view of Capella Columnar's compute-storage separated architecture and how it enables a cluster to be scaled up (or down) to adjust the query performance that is currently deliverable by the cluster. The lower section of Figure 9 shows the storage partitions of a hypothetical eight-partition collection at rest in the service's underlying internal object storage. The upper left section shows a one-node cluster, with all of the collection's storage partitions being assigned to this one compute node, and with the time required to execute a particular query being 2 seconds. The upper right section shows the impact of doubling the cluster's size. Each node of the resulting two-node cluster is now responsible for querying half as many storage partitions, leading to twice the query performance. It is important to note that in reality these numbers are much higher; by default each collection is partitioned

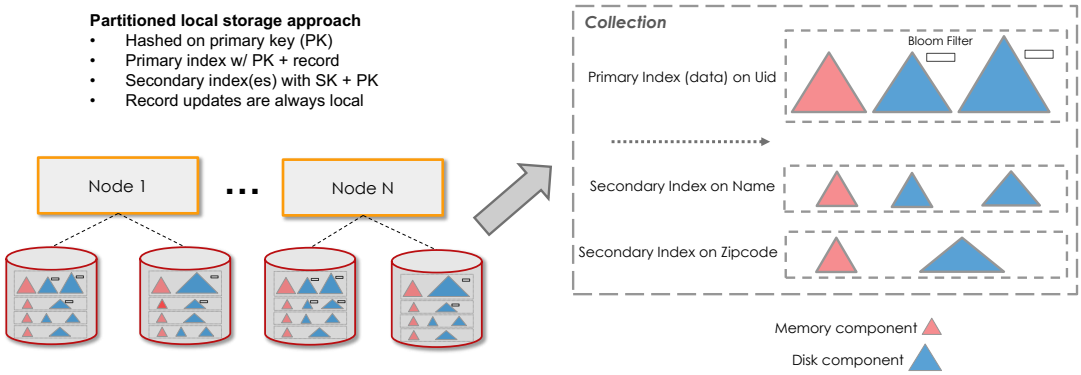

Figure 7: Partitioned Storage and Indexing.

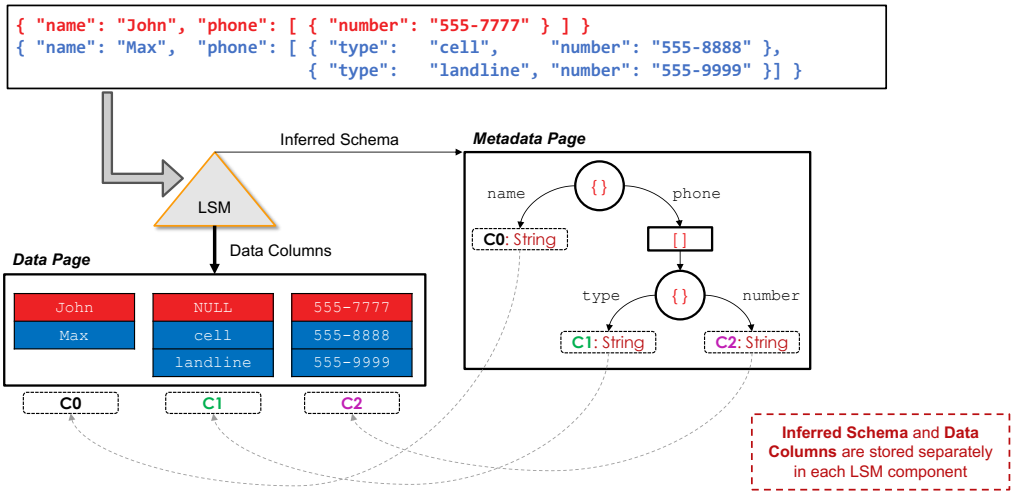

Figure 8: Columnar JSON Storage Format.

into 128 storage partitions to facilitate scaling via parallelism and effective load balancing.

In terms of its database architecture, Columnar is essentially a shared-nothing/shared-disk hybrid system. The data resides in its internal object storage and is accessible by all nodes in a shared-disk-like sense. S3 and GCS are the two flavors of object store that will be used, depending on the cloud platform on which the service is deployed. However, at any given point in time, the system actually runs in shared-nothing mode, with reading and writing from and to a given storage partition being the responsibility of exactly one compute node, as indicated in Figure 9. This responsibility can be reassigned in the event of failures or scaling operations.

Columnar supports virtually unlimited storage by leveraging the cloud object store for remote storage. To minimize latency between compute nodes and remote storage, each node has a local NVMe cache with limited capacity. When it nears capacity, a novel column-oriented eviction policy is triggered. The policy selectively evicts infrequently accessed columns, prioritizing the retention of columns frequently accessed by the workload. If there is no available space in the local cache for hot data, reads and writes occur directly with the remote store. When space becomes available in the local cache (e.g., if a hot data collection is dropped), frequently accessed data will be re-cached when it is next read from remote storage.

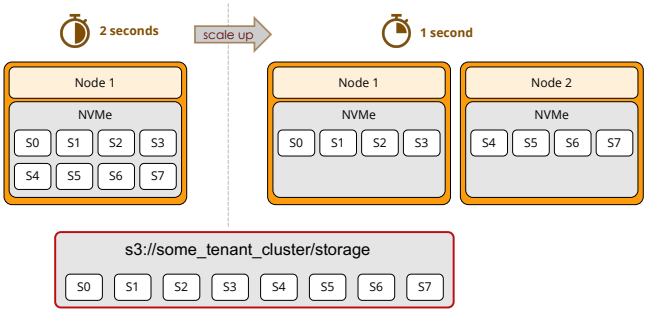

Figure 9: Compute-Storage Separation.

5.3 MPP Query Processing

Query execution in Capella Columnar uses partitioned parallelism, a.k.a. MPP, to execute analytical queries quickly and in a manner that can scale horizontally with the current size of the cluster. Figure 10 shows an example of MPP-based query execution for a GROUP BY aggregate query. In the first phase of query execution the compute nodes perform, in parallel, local grouped aggregation. The results from the first phase are then repartioned by hashing on the grouping key to ensure that all the information for a given group will land on the same node as input to the second phase. Then, in the second phase, the nodes work in parallel to compute the final aggregate values for their assigned groups. More information about parallel query processing and the algorithms used for selection, projection, joins, sorts, aggregation, grouping, and windowing can be found in [26], as the Columnar query engine, like Couchbase Analytics' query engine, is from Apache AsterixDB [3, 15].

5.4 Cost-Based Query Optimization

Capella Columnar's approach to cost-based query optimization (CBO) is based on sampling. For each collection, the query engine also keeps a small, fixed-size random sample of the collection for use in cardinality and cost estimation at query compile time. Users run ANALYZE statements to periodically refresh the samples. When a collection is analyzed, its cardinality is also recorded. Then, to choose an execution strategy for a SQL++ query that involves the collection, its sample's contents, along with its cardinality, provide the information needed by CBO for query planning.

Figure 11 shows this for a simple example query. The query involves two collections, each with various selection predicates, as well as a join predicate. The query optimizer identifies which predicates go with which collection and runs single-collection queries against the samples to determine the numbers of documents that

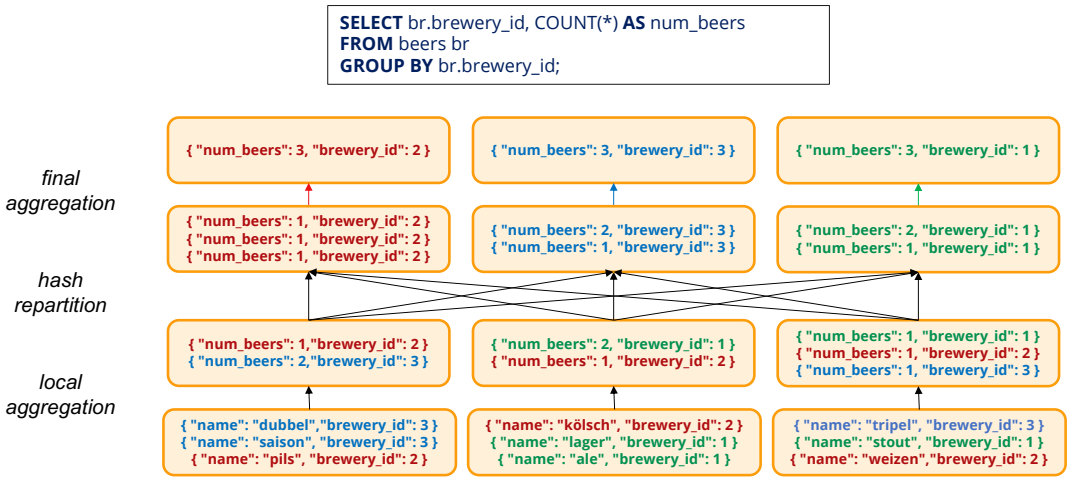

Figure 10: Partitioned Parallelism (MPP).

satisfy their parts of the query criteria. The results are then up-scaled to estimate the cardinalities when the query is run against the actual collections. This allows Columnar CBO to estimate cardinalities for a wide range of otherwise difficult predicates, including LIKE predicates, predicates involving expressions and function calls, and predicates on nested document data. Samples are also used for join optimization, e.g., to estimate the number of distincts in join columns, and when the underlying tables are sufficiently small or a join predicate is complex, their samples are joined. This use of compile-time sampling is unique in the document database world.

5.5 External Collection Access

Figure 12 illustrates the process involved in querying an external collection whose data lives in an external object store rather than being stored and managed by Capella Columnar. The scenario shown involves `myExternalCollection`, an external collection of review data residing in temporally-organized files in object storage. The `CREATE EXTERNAL COLLECTION` statement includes a `PATH` specification that indicates that the organization is a hierarchy involving the year, quarter, and month of the underlying data. The `PATH` specification also indicates the data types involved in the path, and the presence of a parameterized path (or "dynamic path") will cause objects in the collection to appear to have fields `year`, `quarter`, and `month`, each of whose values will be dynamically determined at runtime based on where in the hierarchy an object is found. When a query against this collection is compiled and executed, this path information can be matched against the query's predicates and used to avoid accessing files in unnecessary "folders". We also exploit the scan optimizations available for each data format, and parallel processing is applied when the data from an external collection is being accessed by a query.

5.6 Performance

To provide a brief look at Capella Columnar's performance, we share some results that were obtained from running a mixed noSQL workload benchmark, CH2++ [6, 17], with transactions on a Capella Operational cluster feeding data to a set of analytical queries involving 11 indexed shadow collections on a Capella Columnar cluster. CH2++ is a JSON-ified descendent of CH [22], a relational HTAP benchmark that mixes TPC-C transactions and TPC-H analytics.

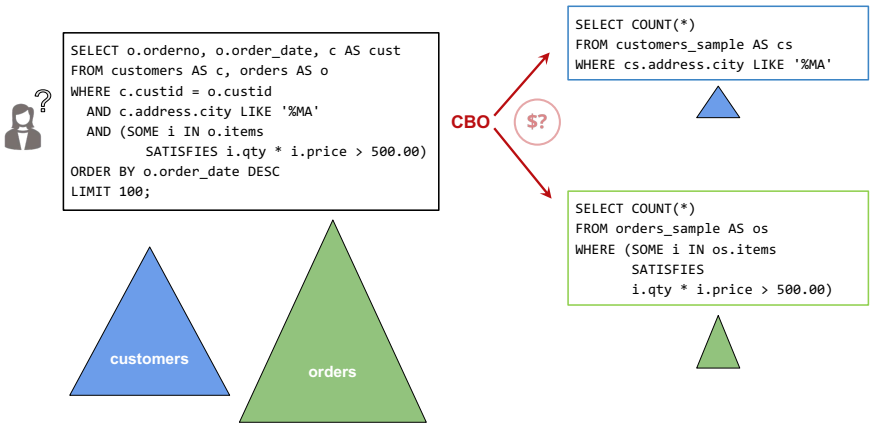

Figure 11: Cost-Based Optimization using Samples.

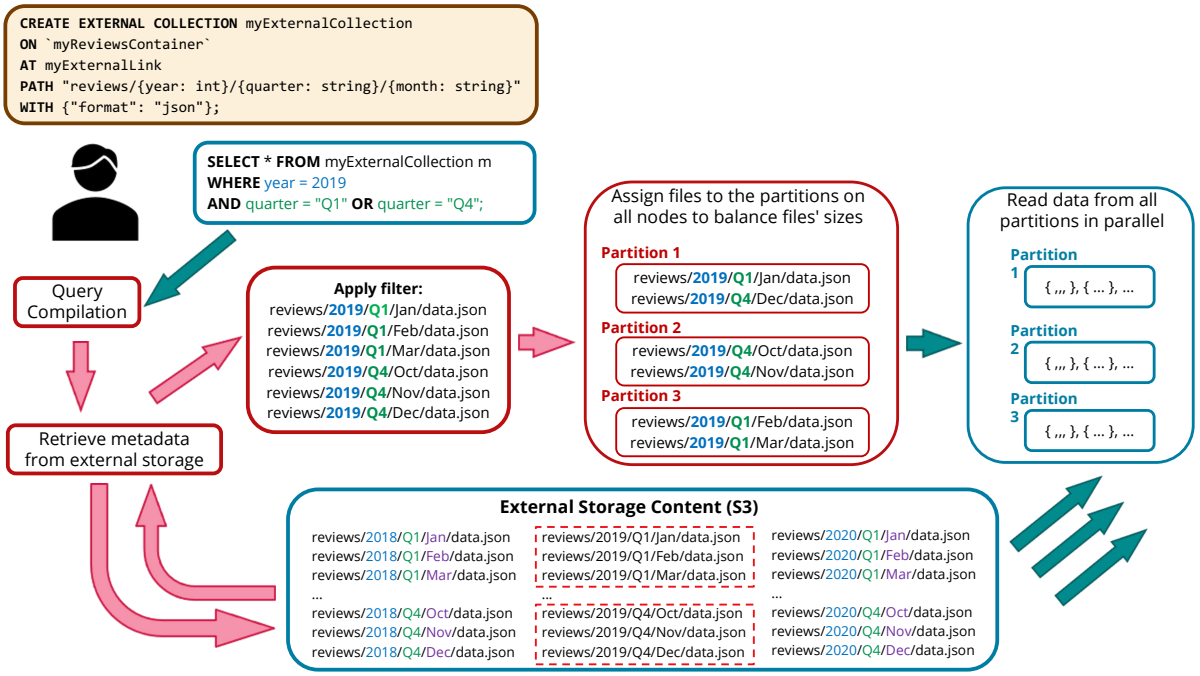

Figure 12: Querying External Collections.

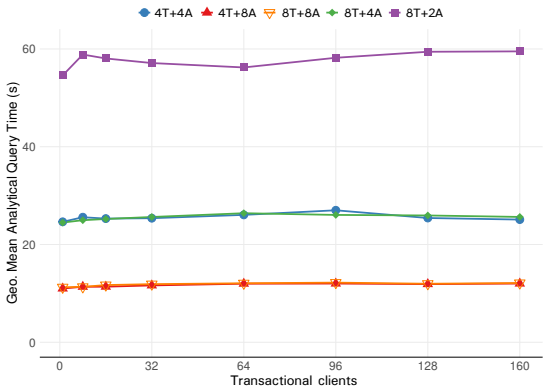

Figure 13: Query Power vs. Operational Load

Figure 13 shows the geometric mean response time (Query Power) for CH2++'s 22 analytical queries with 4 and 8 transactional nodes (4T, 8T) feeding 2, 4, and 8 analytical query nodes (2A, 4A, 8A) using a 1000-warehouse database instance ($\approx 550GB$ of data). We see that query performance scales linearly with the Columnar cluster size. Table 1 shows the benefit of the columnar storage format for collections. Compared to an earlier (and still selectable) row-based format option, query performance is doubled, while ingestion performance (measured while shadow collections were loading) is only slightly slower due to the cost of parsing and shredding the incoming data.

5.7 Differentiation

There are quite a few cloud data warehouses and data lakes today. Redshift [13] and Snowflake [23] are two key warehouse services.

Table 1: Storage Format Performance Impact (4T+4A)

Storage Format	Avg. Data Ingestion Rate	Geo. Mean Query Time	Query Throughput
Column	200.6k obj/sec	26.06 sec	106.85 qry/hr
Row	214.1k obj/sec	55.81 sec	55.72 qry/hr

Both of these warehouse services are relationally focused (i.e., based on SQL, tables, and predefined schemas) and treat JSON as a column type. DeltaLake [12] is a leading data lake service for organizing and analyzing files in object storage. Parquet, its preferred format, supports nesting, but DeltaLake expects file data to have uniform schemas. Atlas [5] is a leading JSON-based cloud service, but it is optimized for transactional use, not analytics. Capella Columnar is unique in its focus on JSON and its treatment of other data models as subsets of JSON. Its schema flexibility allows data to be analyzed in its natural form (i.e., there is no ETL mandate). Columnar allows data to be brought together eagerly and/or lazily from disparate remote systems, in cross-queryable (with SQL++) JSON form.

6 CONCLUSION

In this paper we have presented Couchbase's Capella Columnar service, a service that is uniquely suited to performing large scale JSON data analytics. We covered the kinds of collections that it supports, its SQL++ language, its view and function support, its relational BI tool interoperability, its compute-storage-separation, its columnar storage, its parallel query runtime, its sample-based query optimizer, and its paths for ingesting and querying data from diverse sources. Space has precluded diving into these topics more deeply, but we hope that readers have found this overview valuable.

REFERENCES

[1] Amazon Web Services 2024. *Amazon DynamoDB.* Amazon Web Services. Retrieved November 26, 2024 from https://aws.amazon.com/dynamodb/

[2] Couchbase, Inc. 2025. *About Capella Columnar.* Couchbase, Inc. Retrieved February 2, 2025 from https://docs.couchbase.com/columnar/intro/intro.html

[3] Apache Software Foundation 2025. *Apache AsterixDB.* Apache Software Foundation. Retrieved January 25, 2025 from https://asterixdb.apache.org/index.html

[4] Apache Software Foundation 2025. *Apache Kafka.* Apache Software Foundation. Retrieved March 15, 2025 from https://kafka.apache.org/

[5] MongoDB 2025. *Atlas Database.* MongoDB. Retrieved June 11, 2025 from https://www.mongodb.com/products/platform/atlas-database

[6] Couchbase, Inc. 2025. *CH2++: A Hybrid Operational/Analytical Processing Benchmark for NoSQL Databases.* Couchbase, Inc. Retrieved June 11, 2025 from https://github.com/couchbaselabs/ch2

[7] Couchbase, Inc. 2025. *Couchbase Server.* Couchbase, Inc. Retrieved January 25, 2025 from https://www.couchbase.com/products/server/

[8] 2025. *Introducing JSON.* Retrieved January 25, 2025 from https://www.json.org/json-en.html

[9] MongoDB, Inc. 2025. *Introduction to MongoDB.* MongoDB, Inc. Retrieved January 25, 2025 from https://www.mongodb.com/docs/manual/introduction/#introduction-to-mongodb

[10] Microsoft, Inc. 2025. *Queries in Azure Cosmos DB for NoSQL.* Microsoft, Inc. Retrieved January 25, 2025 from https://learn.microsoft.com/en-us/azure/cosmos-db/nosql/query/

[11] Wail Y. Alkowaileet and Michael J. Carey. 2022. Columnar Formats for Schemaless LSM-based Document Stores. *Proc. VLDB Endow.* 15, 10 (2022), 2085–2097.

[12] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage Over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (Aug. 2020).

[13] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proc. 2022 Int'l. Conf. on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. ACM, New York, NY, USA, 2205–2217.

[14] Dipti Borkar, Ravi Mayuram, Gerald Sangudi, and Michael Carey. 2016. Have Your Data and Query It Too: From Key-Value Caching to Big Data Management. In *Proc. 2016 Int'l. Conf. on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 239–251.

[15] Michael Carey. 2019. AsterixDB Mid-Flight: A Case Study in Building Systems in Academia. In *2019 IEEE 35th Int'l. Conf. on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–12.

[16] Michael Carey, Don Chamberlin, Almann Goo, Kian Win Ong, Yannis Papakonstantinou, Chris Suver, Sitaram Vemulapalli, and Till Westmann. 2024. SQL++: We Can Finally Relax!. In *2024 IEEE 40th Int'l. Conf. on Data Engineering (ICDE)*. 5501–5510.

[17] Michael Carey, Vijay Sarathy, Daniel Nagy, Bo-Chun Wang, Keshav Murthy, M. Muralikrishna, Peeyush Gupta, and Till Westmann. 2025. CH2++: New HOAP for Benchmarking JSON Data Analytics. *TPCTC 2025* (2025).

[18] Rick Cattell. 2011. Scalable SQL and NoSQL Data Stores. *SIGMOD Rec.* 39, 4 (May 2011), 12–27.

[19] Don Chamberlin. 2018. *SQL++ For SQL Users: A Tutorial.* Couchbase, Inc. https://www.couchbase.com/content/analytics/sql-book

[20] Donald Chamberlin. 2024. 50 Years of Queries. *Commun. ACM* 67, 8 (Aug. 2024), 110–121.

[21] Edgar F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387.

[22] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The Mixed Workload CH-benCHmark. In *Proc. Fourth Int'l. Workshop on Testing Database Systems* (Athens, Greece) *(DBTest '11)*. ACM, New York, NY, USA.

[23] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proc. 2016 Int'l. Conf. on Management of Data, SIGMOD Conf. 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 215–226.

[24] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *Proc. 2022 USENIX Annual Technical Conf., USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*. USENIX Association, 1037–1048.

[25] Raman Grover and Michael Carey. 2015. Data Ingestion in AsterixDB. In *Proc. 18th Int'l. Conf. on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015*. OpenProceedings.org, 605–616.

[26] Murtadha Al Hubail, Ali Alsuliman, Michael Blow, Michael Carey, Dmitry Lychagin, Ian Maxon, and Till Westmann. 2019. Couchbase Analytics: NoETL for Scalable NoSQL Data Analysis. *Proc. VLDB Endow.* 12, 12 (Aug 2019), 2275–2286.

[27] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. 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).

[28] Pramod Sadalage and Martin Fowler. 2012. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistnce.* Addison-Wesley, USA.

[29] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around... *SIGMOD Rec.* 53, 2 (July 2024), 21–37.