Query Performance Assessment

Ari Ekmekji

April 2020

This assessment is an inside look into the performance of Rockset, illustrating real-world numbers for what you can expect when querying data in Rockset. For an assessment of query performance, we ingested 100 GB of sample e-commerce booking data and ran a series of queries- 9 in total- that are representative of what a data application might do.

Performance Objectives

Data applications include customer 360s, fraud detection applications, gaming leaderboards and IoT applications. These applications require highly performant, complex queries on fresh data. For this assessment, we used sample datasets and queries that reflect the performance objectives of data applications.

Ability to handle semi-structured data

Application or device data is commonly available to developers in datasets that do not have a fixed schema. For example, JSON data regularly contains deeply nested arrays and objects, mixed data types, null values and missing fields. Time-consuming schema definition and data prep is traditionally required before a developer can read the data, elongating the application development lifecycle. We ingested deeply nested JSON documents with sparse fields for the assessment without flattening the data ahead of time.

Rockset permits schemaless ingest of data, making it easy to write semi-structured data into the system. Rockset uses Converged Indexing™ to automatically index every field of the ingested data in a row-based store, column-based store, and a search index. With indexing, sparse fields do not require scanning an entire column in order to run computation on the few, non-null rows. Instead, Rockset identifies those rows directly every time and can process them in a fraction of the time most databases require.

Fast performance for data application queries

Data applications require complex queries- aggregations, joins, filters- on data from different sources. They also generally involve selective predicates or querying a subset of the data based on the user, product line, geo, etc. which differ from traditional data warehouse queries. Many data backends do not support complex queries, specifically joins, so developers are forced to either pre-join the data or write extensive custom code to do application-side joins, both of which increase the development time of applications and, potentially, the query latency.

Because of Converged Indexing, every query can make use of an index, so execution time depends solely on the number of documents matched by the predicates and never the entire collection size. Even though the datasets used in the assessment have millions of rows, with aggregations and joins, Rockset is able to achieve a query latency of 10s of milliseconds on queries with selective predicates.

Rockset built a schemaless SQL query engine from the ground up as traditional SQL systems do not handle mixed types or deal well with deeply nested data. Rockset supports ANSI SQL- including joins, filters, and aggregations natively- and has added extensions, including an UNNEST function, that can be used to expand arrays of values to be queried.

Serverless auto-scaling for out-of-the-box query performance

Traditional databases require administrators to spin up and manage infrastructure behind the scenes to achieve performance over constantly changing dataset sizes and query patterns. Databases with serverless auto-scaling remove the need to manage the complexity of the system such as manually tuning performance for concurrency, data scaling and resource-intensive queries.

With Rockset, developers can move their applications from development to production faster as they are no longer bottlenecked on infrastructure management.

The Assessment

The dataset

We used two nested JSON datasets that are modeled on an e-commerce reservation system to showcase Rockset's ability to handle semi-structured datasets. The datasets are publicly available at: s3://rockset-public-datasets/reservations.

Reservation data: 100GB in size with 300M rows of reservation data that mirrors real-world, deeply nested application or device data. The reservation data lists resources that are booked including the start and end dates of the booking, total number of days booked, the promotion code used in the booking (if any), the price, and user rating. Each reservation is for a generic resource- which could represent a hotel room, dinner reservation, flight, etc. in real life.

User data: 100MB in size with 1M rows of user data that mirrors meta-data that is used to provide necessary contextual information. The user data includes the name, birthdate, and ID of users. In this example, a user can both book a reservation and create a reservation.

While we only ingested 100 GB for this assessment, most of our customers use Rockset to build real-time applications on TBs of data. You can explore the data applications built on Rockset on the customers page.

Reservation schema

```
{
    "price": {
        "promo": {
            "code": "DOORBUSTER",
            "discount": 88.7
        },
        "total": 298.93,
        "subtotal": 354.82,
        "tax": 32.82
    },
    "bookingUser":
"ac441d2b-9094-4b33-881b-fcd24623a4a4",
    "resourceId":
"0c003030-6533-4659-998e-70f6dcda5b50",
    "end": "2019-08-29",
    "start": "2019-08-25",
    "listingUser": "1e72440f-a2d6-40ef-
ae5f-cf1c41e66b04",
    "_id": "6d0dea93-6a34-4091-
a2c8-23ffbbc7eae9",
    "rating": 4,
    "days": 4,
    "reviewText": "Velit ipsum voluptatem
sit neque dolorem aliquam consectetur."
}
```

1 The reservations.price.promo is optional in the booking system. The field exists for some records, ~1%, but is NULL for the rest.

2 This data is nested- you can see the reservation data has 3 levels of nesting and the user data has 2 levels of nesting

Users schema

```
{
    "name": {
        "last": "Hess",
        "first": "John"
    },
    "_id":
"088dc96d-81d8-40d9-98d3-53d75ed47dce",
    "verified": true,
    "birthday": "1994-09-22"
}
```

The queries

As stated earlier, the queries were drawn from patterns we see in data application queries including joining large collections with smaller dimension collections, low-cardinality aggregations within a large collection, operations on nested fields/arrays, and selective predicates over large collections.

We asked the following questions of the data:

1. What were the total number of reservation days booked for specific users?
2. How frequently were promotion codes used in reservation bookings?
3. What was the average discount received when a specific promotion code was applied?
4. Which user booked the longest reservation for a specific set of resources?
5. For a specific user, what was the average rating they gave to each resource?
6. Did a user book overlapping reservations? If so, what were the reservations that overlapped?
7. Did a resource have overlapping reservations? If so, what were the reservations that overlapped?
8. For a specific user, which promotion codes did they use?
9. For a specific resource, what were the last 10 reservation bookings? Which users booked the reservations?

Rockset's Configuration

Rockset's configuration

We used Rockset's out-of-the-box configuration to run these queries to assess query performance, meaning we did not hand tune the cluster. With strong out-of-the-box performance, developers can build data applications at scale faster. The performance assessment used the same configuration that we provide to every trial account: Rockset's c12 instance type and 2 replicas for consistency and availability. You can find more information on this configuration on our pricing page.

How we ran the assessment

As we intended for this assessment to mirror how an application developer would use Rockset, we used the Python client to ingest data, create the collections, and issue the queries.

We provided the configuration files to load the collections into Rockset from the Python client below. You can think of collections in Rockset as tables in the relational world. You can see from the reservation and user collections that we used field mappings to specify transformations for time series indexing (since JSON can't natively encode dates).

You can access them from: rock create -f users.yaml and rock create -f reservations.yaml.

<u>Reservations Collection:</u>

```
workspace: commons
name: reservations
sources:
- s3:
    bucket: rockset-public-datasets
    prefix: reservations/reservations/
type: COLLECTION
field_mappings:
- name: start
  input_fields:
  - field_name: start
    if_missing: SKIP
    is_drop: false
    param: param
  output_field:
    field_name: start
    on_error: FAIL
    value:
      sql: CAST(:param AS DATE)
- name: end
  input_fields:
  - field_name: end
    if_missing: SKIP
    is_drop: false
    param: param
  output_field:
    field_name: end
    on_error: FAIL
    value:
      sql: CAST(:param AS DATE)
```

Users Collection:

```
workspace: commons
name: users
sources:
- s3:
    bucket: rockset-public-datasets
    prefix: reservations/users
type: COLLECTION
field_mappings:
- name: birthday
  input_fields:
  - field_name: birthday
    if_missing: SKIP
    is_drop: false
    param: param
  output_field:
    field_name: birthday
    on_error: FAIL
    value:
      sql: CAST(:param AS DATE)
```

Results

Results Table

Query ID	Description	Rows Processed	Runtime (ms)	Rows Returned	Join Type
1	Total days booked for some specific users	591	25	2	Hash Join
2	Frequency of all promo codes	3,000,994	1240	12	N/A
3	Usage of a particular promo code	11,346	23	1	N/A
4	Who had the longest reservation for participating resources	146	25	73	Lookup Join
5	Avg review by resource for a user	274	21	9	N/A
6	Find overlapping reservations for a user	331	158	318	Self Hash Join
7	Find overlapping reservations for a resource	40	29	1	Self Hash Join
8	Usage of promos by a given user	9	290	7	N/A
9	Last 10 users to have a reservation for a resource	52	21	10	Lookup Join

Performance of individual queries

For each query we provide the SQL and the output of the EXPLAIN command for that query.

Query 1

Query: What were the total number of reservation days booked for specific users?

```
-- Total days booked for some specific users
SELECT
    CONCAT(u.name.first, ' ', u.name.last) name,
    q."days"
FROM
    (
        SELECT
            r.bookingUser,
            SUM(r."days") "days"
        FROM
            commons.reservations r
        WHERE
            r.bookingUser IN (
                'b735b18a-50bd-4d5e-8d71-acf88aa95150',
                'f22c49aa-7a41-43b3-86e8-5fe67ee8297b'
            )
        GROUP BY
            r.bookingUser
    ) q
    INNER JOIN commons.users u ON q.bookingUser = u._id
WHERE
    u._id IN (
        'b735b18a-50bd-4d5e-8d71-acf88aa95150',
        'f22c49aa-7a41-43b3-86e8-5fe67ee8297b'
    )
```

Results: The query runtime was 25ms and processed 591 rows. The query used an aggregation to sum the total days booked, a hash join to display both user and reservation data, and a selective predicate to filter on a specific set of users.

Explain:

```
select name:$7, days:$3
  project $7=(concat($5, " ", $6))
    hash inner join on ($1 = $4)
      aggregate on ($1): $3=sum($2)
        index filter on commons.reservations:
fields($1=bookingUser, $2=days),
query($1:string["b735b18a-50bd-4d5e-8d71-
acf88aa95150","b735b18a-50bd-4d5e-8d71-acf88aa95150"],
string["f22c49aa-7a41-43b3-86e8-5fe67ee8297b","f22c49aa-7a41-43
b3-86e8-5fe67ee8297b"])
      index filter on commons.users: fields($4=_id,
$5=name.first, $6=name.last),
query($4:string["b735b18a-50bd-4d5e-8d71-
acf88aa95150","b735b18a-50bd-4d5e-8d71-acf88aa95150"],
string["f22c49aa-7a41-43b3-86e8-5fe67ee8297b","f22c49aa-7a41-43
b3-86e8-5fe67ee8297b"])
```

Query 2

Query: How frequently were promotion codes used in reservation bookings?

```sql
-- Frequency of all promo codes
SELECT
    r.price.promo.code,
    COUNT(*) "count"
FROM
    commons.reservations r
WHERE
    r.price.promo.code IS NOT NULL
GROUP BY
    r.price.promo.code
```

Results: The query runtime was 1,240ms and processed 3,000,994 rows. The field price.promo.code was applied to ~1% of the rows to display Rockset's ability to handle sparse fields. The query also used a low-cardinality group-by to display the results by individual promotion codes.

Explain:

```
select code:$1, count:$2
  aggregate on ($1): $2=count_star_()
    index filter on commons.reservations:
fields($1=price.promo.code), query($1:array, bool,
bytes[H"",Maxbytes), date[1970-01-01,2147483647-12-31],
datetime[1970-01-01T00:00:00.000000,2147483647-12-31T23:59:59.9
99999], float[-inf,inf],
geography[RANGE_ENDPOINT(),RANGE_ENDPOINT(~)],
int[-9223372036854775808,9223372036854775807], object,
string["",Maxstring), time[00:00:00.000000,23:59:59.999999],
timestamp[@-9223372036855.224192,@9223372036854.775807])
```

Query 3

Query: What was the average discount received when a specific promotion code was applied?

```sql
-- Usage of a particular promo code
SELECT
    AVG(r.price.promo.discount)
FROM
    commons.reservations r
WHERE
    r.price.promo.code = 'BLOWOUT'
```

Results: The query runtime was 23ms and processed 11,346 rows. The query used a highly selective predicate, a single promotion code, for fast performance.

Explain:

```
select "?AVG":$3
  aggregate on (): $3=avg($2)
    index filter on commons.reservations:
fields($1=price.promo.code, $2=price.promo.discount),
query($1:string["BLOWOUT","BLOWOUT"])
```

Query 4

Query: Which user booked the longest reservation for a specific set of resources?

```
-- Who had the longest reservation for participating resources
SELECT
    CONCAT(u.name.first, ' ', u.name.last) name,
    r."days"
FROM
    commons.users u
    INNER JOIN commons.reservations r ON r.bookingUser = u._id
HINT(join_strategy = lookup)
WHERE
    r.resourceId IN (
        '78695ac9-c9ce-490e-84d8-af45ff51137c',
        '0b3b34cd-52ef-40fb-a7a0-e03eea4aa076',
        'a0fed2d5-a393-4689-8b70-543cc50ecb39'
    )
ORDER BY
    "days" DESC
```

Result: The query runtime was 25ms and processed 146 rows. We used Rockset's HINT functionality to provide guidance to the query optimizer on the correct join strategy to use for this query. The lookup join was used because we are joining on a limited number of rows, a specific set of resource IDs, for this query.

Explain:

```
select name:$7, days:$4
  project $7=(concat($5, " ", $6))
    sort $4 desc
      lookup join with commons.users on ($1 = $2)),
fields($2=_id, $5=name.first, $6=name.last), query(all)
        index filter on commons.reservations:
fields($1=bookingUser, $4=days, $3=resourceId),
query($3:string["0b3b34cd-52ef-40fb-a7a0-
e03eea4aa076","0b3b34cd-52ef-40fb-a7a0-e03eea4aa076"],
string["78695ac9-c9ce-490e-84d8-af45ff51137c","78695ac9-
c9ce-490e-84d8-af45ff51137c"], string["a0fed2d5-
a393-4689-8b70-543cc50ecb39","a0fed2d5-
a393-4689-8b70-543cc50ecb39"])
```

Query 5

Query: For a specific user, what was the average rating they gave to each resource?

```
-- Avg review by resource for a user
SELECT
    r.resourceId,
    count(*),
    AVG(r.rating)
FROM
    commons.reservations r
WHERE
    r.listingUser = '9fd7d081-8fcb-4bb6-aa55-8e278087d3f9'
GROUP BY
    r.resourceId
```

Result: The query runtime was 21ms and processed 274 rows. This was a query on a single collection that filtered on a particular user.

Explain:

```
select resourceId:$2, "?count":$5, "?AVG":$4
  aggregate on ($2): $4=avg($3), $5=count_star_()
    index filter on commons.reservations:
fields($1=listingUser, $3=rating, $2=resourceId),
query($1:string["9fd7d081-8fcb-4bb6-
aa55-8e278087d3f9","9fd7d081-8fcb-4bb6-aa55-8e278087d3f9"])
```

Query 6

Query: Did a user book overlapping reservations? If so, what were the reservations that overlapped?

```sql
-- Find overlapping reservations for a user
WITH bookings AS (
    SELECT
        _id,
        r.start start,
        r."end"
    FROM
        commons.reservations r
    WHERE
        r.bookingUser = 'f306c0d5-98b9-4289-8b16-5124b15b0987'
)
SELECT
    b1._id,
    COUNT(*)
FROM
    bookings b1
    INNER JOIN bookings b2 ON b2.start >= b1.start
    AND b2.start <= b1."end"
    AND b1._id != b2._id
GROUP BY
    1
```

Results: The query runtime was 158ms and processed 331 rows. The query used a hash join and aggregation.

Explain:

```
select _id:$8, "?COUNT":$13
  aggregate on ($8): $13=count_star_()
    hash inner join on (); extra($11 <= $6, $11 >= $7, $8 !=
$12)
      index filter on commons.reservations: fields($6="end",
$8=_id, $5=bookingUser, $7=start),
query($5:string["f306c0d5-98b9-4289-8b16-5124b15b0987","f306c0d
5-98b9-4289-8b16-5124b15b0987"])
      index filter on commons.reservations: fields($12=_id,
$9=bookingUser, $11=start),
query($9:string["f306c0d5-98b9-4289-8b16-5124b15b0987","f306c0d
5-98b9-4289-8b16-5124b15b0987"])
```

Query 7

Query: Did a resource have overlapping reservations? If so, what were the reservations that overlapped?

```
-- Find overlapping reservations for a resource
WITH bookings AS (
    SELECT
        r._id,
        r.start start,
        r."end"
    FROM
        commons.reservations r
    WHERE
        r.resourceId = '336ce95d-1800-4605-a5ff-9d97a1011643'
)
SELECT
    COUNT(*)
FROM
    bookings b1
    INNER JOIN bookings b2 ON b2.start >= b1.start
    AND b2.start <= b1."end"
    AND b1._id != b2._id
```

Results: The query runtime was 29 ms and processed 40 rows. The query used a self hash join.

Explain:

```
select "?COUNT":$13
  aggregate on (): $13=count_star_()
    hash inner join on (); extra($11 <= $6, $11 >= $7, $8 !=
$12)
      index filter on commons.reservations: fields($6="end",
$8=_id, $5=resourceId, $7=start),
query($5:string["336ce95d-1800-4605-
a5ff-9d97a1011643","336ce95d-1800-4605-a5ff-9d97a1011643"])
      index filter on commons.reservations: fields($12=_id,
$9=resourceId, $11=start), query($9:string["336ce95d-1800-4605-
a5ff-9d97a1011643","336ce95d-1800-4605-a5ff-9d97a1011643"])
    AND b1._id != b2._id
```

Query 8

Query: For a specific user, which promotion codes did they use?

```sql
-- Usage of promos by a given user
SELECT
    r.price.promo.code AS promo_code,
    COUNT(*) AS promo_booking_count,
    SUM(r.price.promo.discount) AS total_saved,
    r.bookingUser
FROM
    reservations r
WHERE
    r.bookingUser = '053cfb0d-0454-4ad1-b670-db718f1ded50'
    AND r.price.promo.code IS NOT NULL
GROUP BY
    1,
    4
ORDER BY
    3 DESC
```

Results: The query runtime was 290 ms and processed 9 rows. The query included aggregations, order by, and multiple predicates.

Explain:

```
select promo_code:$1, promo_booking_count:$5, total_saved:$4,
bookingUser:$2
  sort $4 desc
    aggregate on ($1, $2): $5=count_star_(), $4=sum($3)
      index filter on commons.reservations:
fields($2=bookingUser, $1=price.promo.code,
$3=price.promo.discount), query(and($1:array, bool,
bytes[H"",Maxbytes), date[1970-01-01,2147483647-12-31],
datetime[1970-01-01T00:00:00.000000,2147483647-12-31T23:59:59.9
99999], float[-inf,inf],
geography[RANGE_ENDPOINT(),RANGE_ENDPOINT(~)],
int[-9223372036854775808,9223372036854775807], object,
string["",Maxstring), time[00:00:00.000000,23:59:59.999999],
timestamp[@-9223372036855.224192,@9223372036854.775807],
$2:string["053cfb0d-0454-4ad1-b670-
db718f1ded50","053cfb0d-0454-4ad1-b670-db718f1ded50"]))
```

Query 9

Query: For a specific resource, what were the last 10 reservation bookings? Which users booked?

```
-- Last 10 users to have a reservation for a resource
SELECT
    CONCAT(u.name.first, ' ', u.name.last) name,
    r."days" length_of_stay
FROM
    users u
    INNER JOIN reservations r ON r.bookingUser = u._id
HINT(join_strategy = lookup)
WHERE
    r.resourceId = 'eb69b13a-6f65-4d01-8464-be0e9a48073d'
ORDER BY
    r.start DESC
LIMIT
    10
```

Results: The query runtime was 21 ms and processed 52 rows. The query used a lookup join and an order by statement.

Explain:

```
select name:$8, length_of_stay:$4
  limit 10 ordered by $7 desc
    project $8=(concat($5, " ", $6))
      sort $7 desc
        lookup join with commons.users on ($1 = $2)),
fields($2=_id, $5=name.first, $6=name.last), query(all)

          index filter on commons.reservations:
fields($1=bookingUser, $4=days, $3=resourceId, $7=start),
query($3:string["eb69b13a-6f65-4d01-8464-
be0e9a48073d","eb69b13a-6f65-4d01-8464-be0e9a48073d"])
```

Summary

The goal of the query assessment was to provide a frame of reference to evaluate Rockset as a real-time database in the cloud. As displayed in the results, Rockset runs complex queries on semi-structured data in milliseconds without any manual performance tuning. Queries that used selective predicates were able to return in 10s of milliseconds, making Rockset a great fit for data applications that automate intelligent decisions on real-time data.

Rockset attained a high level of performance because of its underlying tech, including schemaless ingest of semi-structured data, Converged Indexing™ for fast query performance, and serverless auto-scaling for out-of-the-box performance at scale.

Next Steps

Sign Up Sign up for a free 14 day trial of Rockset and receive $300 in free credits to reproduce the results. Start my free trial>>

Join the community Join the Rockset slack community. Join now>>

Connect with Us Connect with a solution architect Connect now>>