CLOUDERA

Efficient handling of geometry data in Apache Impala with Parquet files

Csaba Ringhofer
Daniel Becker

What is Apache Impala?

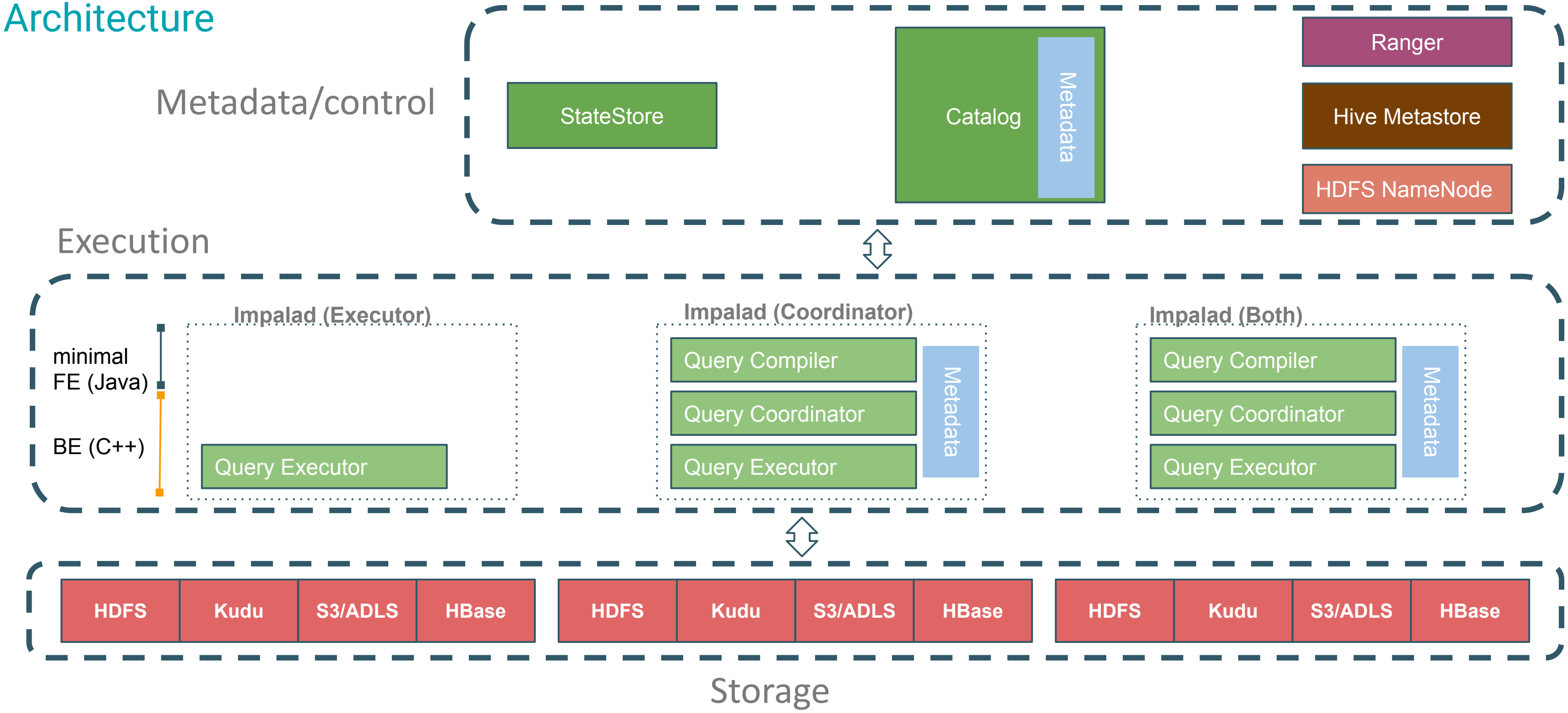
- Distributed, massively parallel SQL database engine
- Originally designed for Hadoop
- Main feature is speed
  - backend (distributed query execution) is written in C++
    - uses LLVM runtime code generation
  - frontend (query planning, optimisation) is in Java

What is Apache Impala?

- Supports various storage systems
  - HDFS, Ozone
  - S3, ADLS
  - Kudu, HBase etc.
- Table formats
  - Hive
  - Iceberg
- File formats
  - Parquet, ORC, text etc.

What is Apache Impala?

Architecture



Metadata/control

StateStore

Catalog

Metadata

Ranger

Hive Metastore

HDFS NameNode

Execution

minimal FE (Java)

BE (C++)

Impalad (Executor)

Query Executor

Impalad (Coordinator)

Query Compiler

Query Coordinator

Query Executor

Metadata

Impalad (Both)

Query Compiler

Query Coordinator

Query Executor

Metadata

HDFS | Kudu | S3/ADLS | HBase | HDFS | Kudu | S3/ADLS | HBase | HDFS | Kudu | S3/ADLS | HBase

Storage

CLOUDERA

Status of geospatial features in Impala

Summary

Already exists:

- Large number of geospatial functions
  - Java functions (shared with Apache Hive)
    - originally from Esri Spatial Framework for Hadoop
- Test coverage is basic - > not yet officially supported

In progress:

- Porting functions to c++
- Looking for file format as recommended storage

Planned:

- Extend test suite to become supported

Status of geospatial features in Impala

Limitations

- Mostly 2D geometry support
  - only 1 geography function
  - limited 3d / 4d support (Z/M)
- 6 types of geometry are available:
  - POINT / LINESTRING / POLYGON (+ MULTI versions)
- No dedicated GEOMETRY data type, BINARY is used instead
- Only a set of functions
  - no expression rewrites
  - no advanced algorithms like geospatial join

Status of geospatial features in Impala

Recent improvements

- Geospatial functions originally implemented in Java
  - slower than C++
  - C++ code (Impala backend) has to call into Java code for each row, huge overhead
- Reimplemented some of the most important functions in C++
  - 26 out of ~140, including **st_intersect()**
  - using `boost::geometry`
  - results are binary compatible with the Java version
  - 40-50x speedup in some cases

What table format to use for geospatial data?

- Hive table format
  - files of a table stored in a file system directory
  - partitions in subdirectories
- Iceberg table
  - files of a table (and partitions) stored in metadata files
  - file level min/max stats in metadata
    - available at planning
    - can be used for bounding rectangle check
- Kudu table - out of scope of presentation
  - GeoMesa had a solution

What file format to use for geospatial data?

Considerations

Different use cases:

- Should work well with different file systems:
  - Hadoop (HDFS or Ozone)
  - object stores (S3, ABFS …)
- Efficient handling of different WHERE filters:
  - select all - no filtering
  - predicate on geometry column
  - predicate on non-geometry columns
- SELECT * vs SELECT subset of columns (projection)

CLOUDERA

What file format to use for geospatial data?

IO considerations

Hadoop (HDFS):

- Files stored as 1 or more large blocks
- Blocks are present on 1 or more hosts (replicas)
- Reading data from local blocks is much faster
- Splittable file formats are preferred:
  - schedule local blocks to hosts
  - minimise data read from remote blocks

What file format to use for geospatial data?

IO considerations

Object stores:

- All data is remote
- Data caching is critical for performance
  - Impala caches data to both memory and disk
  - files have “host affinity” to improve caching

What file format to use for geospatial data?

	Already supported in Impala	Efficient	Splittable	Supported in Iceberg
CSV	YES	NO	YES	NO
Parquet	YES	YES	YES	YES
ORC	YES (read-only)	YES	YES	YES
Shapefile	NO	YES	NO	NO
SpatiaLite	NO	YES	NO	NO
GeoJson / EsriJson	NO	NO	NO	NO

What file format to use for geospatial data?

File formats already supported by Impala

CSV

- Geometries stored as
  - x/y (points) or
  - WKT (Well-Known-Text) or
  - hex/base64 WKB (Well-Known-Binary)
- Pros:
  - can already be read by Impala
  - many tools can export to it
- Cons:
  - generally inefficient
  - no indexing

GeoJson, EsriJson

- Similarly inefficient as CSV, not supported by Impala yet

What file format to use for geospatial data?

File formats already supported by Impala

Parquet

- Parquet is the file format most efficiently read by Impala
- Pros:
  - very fast scanner in Impala
  - min/max filters can be used for indexing
  - columnar encoding/compression can store attributes efficiently
- Cons:
  - does not seem to be commonly used in the geospatial world

ORC

- Mostly the same as Parquet

What file format to use for geospatial data?

File formats already supported by Impala

ORC

- Pros:
  - mostly the same as for Parquet
  - could be used with Hive Full ACID tables
- Cons:
  - no known geospatial support
  - read-only in Impala

What file format to use for geospatial data?

File formats not yet supported by Impala

Shapefile

- Geospatial vector data format for geographic information system (GIS) software
- Developed by Esri
- Can describe points, lines, and polygons (+ MULTI versions)
- Pros:
    - commonly used, many tools support it
    - the geospatial functions in Impala use this format in memory
        - scanning could be potentially efficient
- Cons:
    - not a single file but a collection of files for a dataset
        - scanner would need to be extended to read multiple files together
        - splitting can be problematic

What file format to use for geospatial data?

File formats not yet supported by Impala

SpatiaLite

- SQLite db file that can contain several geometries
- Used both as a full geospatial db (like a local PostGis) and as an interchange format for single features
- Pros:
  - commonly used
  - SQLite has mature libraries
  - has indexes, point lookup could be fast
  - could be used for non-geospatial data too
- Cons:
  - using a pack of SQLite dbs as a db format looks like a hack
  - bulk read/write would probably be slow compared to Parquet

What file format to use for geospatial data?

File formats not yet supported by Impala

GeoJson / EsriJson

- Based on JSON
- Supports points, line strings, polygons and multi-part collections of these types
- Pros:
  - commonly used
  - the ESRI Hive framework contains a SERDE
- Cons:
  - generally inefficient (even more than CSV)
  - no indexing
  - splitting is problematic

Parquet deep dive

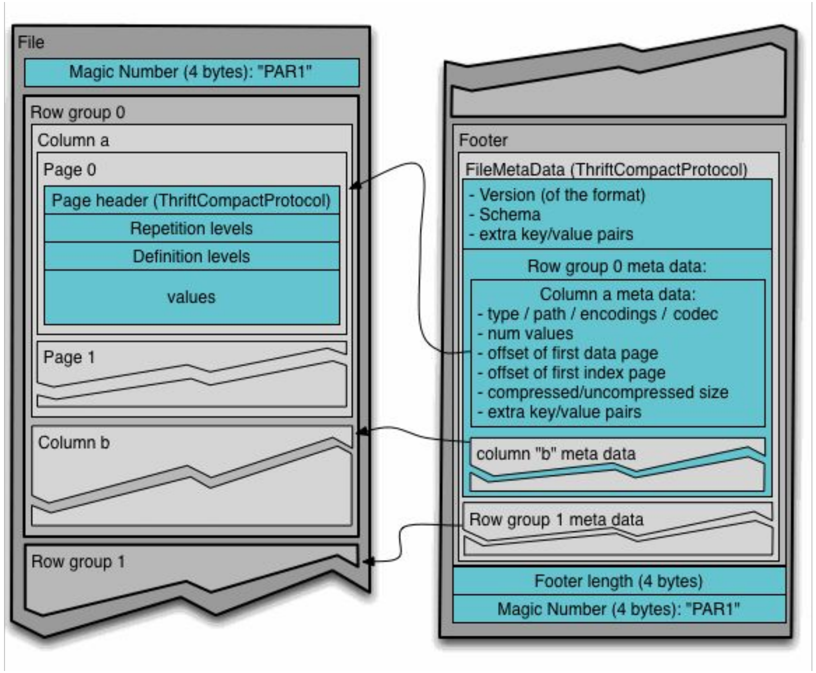
Introduction

- A compressed, efficient columnar data representation originally for the Hadoop ecosystem
- Supports complex nested data structures based on the Dremel paper
  - repetition levels, definition levels
- Supports various compressions and encodings
  - compression on a per-column level
  - encoding on a per-page level

Parquet deep dive

File format

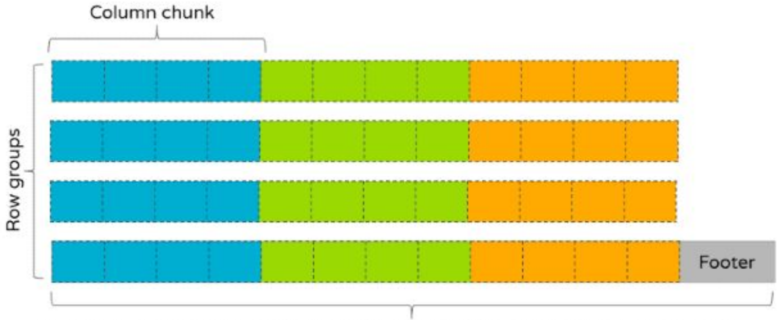
- Rows divided into row groups
- Values stored in a column-oriented way
- Column chunk: the part of a column that is in a single row group
  - may consist of multiple pages
- Each page has its own encoding
- File metadata is at the end (footer) to allow single pass writing



Parquet deep dive

File format

- Rows divided into row groups
- Values stored in a column-oriented way
- Column chunk: the part of a column that is in a single row group
    - may consist of multiple pages
- Each page has its own encoding
- File metadata is at the end (footer) to allow single pass writing



Row groups are aligned HDFS block size, and they are the units of parallelization.

Parquet deep dive

Filtering techniques

Column index

- Statistical information on the values of a column in a row group
- Contains *min* and *max* values for each page of a column
- We can skip pages that contain no values that satisfy the predicates
  - for ordered columns we can use binary search
- Example:

  `SELECT id FROM tbl WHERE id >= 5;`
  - we can skip pages where the *max* value is less than 5

Parquet deep dive

Filtering techniques

Dictionary filtering

- Dictionary encoding
  - store all the values that occur in a column chunk in a dictionary page
  - in subsequent data pages, only store indices into the dictionary
  - useful if the number of distinct values (NDV) is small
    - if NDV is too large, we can use Bloom filters
- We can skip column chunks if the values in the dictionary do not satisfy the predicates
- Example:
  `SELECT transaction_id WHERE customer_id = 125;`
  - we can skip the column chunk if the dictionary does not contain the value 125

Parquet deep dive

Filtering techniques

Bloom filtering

- Probabilistic data structure
  - if a value was inserted into the filter, a check returns `true`
  - if a value was not inserted, a check returns `false` with high probability (may also return `true`)
  - Less precise than a dictionary but can be used with higher NDV
- Example:
  `SELECT transaction_id WHERE customer_id = 125;`
  - we can skip the column chunk if the Bloom filter returns `false` for the value 125

Parquet deep dive

Filtering techniques

Lazy materialisation

- In a query where multiple columns are retrieved, first read and materialise the columns that are involved in predicates
- Evaluate the predicates
- Only materialise the remaining columns for the rows that survive (i.e. are not discarded by the predicates)
- Example:
  `SELECT transaction_id WHERE customer_id = 125;`
  - we only read `transaction_id` for the rows where `customer_id` is `125`
-

Parquet deep dive

Libraries

Different Parquet libraries may read/write files differently!

- Java: parquet-mr
- C++: parquet-cpp (moved to Apache Arrow)
- Impala has its own C++ Parquet scanner
- Python: pyarrow.parquet / fastparquet

Many parameters to fine-tune writing.

Parquet with geospatial data

Existing solution: GeoParquet

GeoParquet provides a standard geospatial representation in Parquet

- Actively developed, this slide is based on v1 specification!
- Stores geometries as BINARY columns (byte array)
  - Using WKB (well-known binary) format
- Adds JSON metadata to the Parquet row group header
- File level bounding box for filtering
- Already supported by several libraries

Parquet with geospatial data

GeoParquet - why not practical for Impala (yet)?

File vs table level format

- GeoParquet adds metadata at file level
- Impala needs table level metadata
- Per file variability would complicate query planning

Not optimal for Impala

- needs WKB -> shape conversion during reading
- Impala uses shapefile’s binary format in memory
- no page level indexing

Parquet with geospatial data

Point data

WKB (or other binary) vs 2 double columns?

lat/lon double column pair is much more efficient!

- Smaller size: no extra fields and length stored
- Decoding can be skipped for simple filters
- Allows min-max filters with existing Parquet libraries

Page level indexing is possible if pages contain “nearby” points

- Sorting the rows during insert can achieve this (e.g. z-order)

Parquet with geospatial data

Point data - partitioning

Point data can be easily partitioned by dividing space into cells

- e.g geohash at some resolution level
- cell size is critical
  - to large: ineffective partitioning pruning
  - too small: over partitioning, small file problem
- Iceberg tables: lat/lon min/max filters can be applied during planning
- Hive tables:
  - query rewrite needed prune partitions during planning
  - lat/lon min/max filter can be applied during execution

CLOUDERA

Parquet with geospatial data

Point data - sorting

How to get finer filtering than partitioning?

Sort points during insert using a space filling curve

- Groups “nearby” points together
- Improves file level filtering within partition
- Allows page level filtering

Parquet with geospatial data

Point data - adding cell_id column

Cell id of point can be added as separate column

- Smaller cell size than during partitioning
- Goal: small NDV (number of distinct values) per file
  - low NDV -> very efficient encoding, minimal overhead
  - cell_id can be used for filtering directly
- Query can be rewritten to also filter on cell_id
- Useful only if number of intersected cells is small

Parquet with geospatial data

Point data - adding cell_id column: benefits

Query can be rewritten to also filter on cell_id

- Derive = or IN filter from bounding box

```
WHERE lat <= … AND lon <= …
    ->
WHERE cell_id IN (<list if intersected cells>)
  AND lat <= … AND lon <= …
```

- Allows dictionary and bloom filtering on cell_id

Parquet with geospatial data

Point data - adding cell_id column: overhead

Low NDV allows dictionary encoding in Parquet

- storage cost: log(NDV) bits per row

RLE (run length encoding) is used for repeated elements

- very efficient encoding for sorted and low NDV data
- Theoretical storage cost: NDV * log(row_count)
- Example: NDV(cell_id) ~= 4K:

lat, lon	cell_id (unsorted)	cell_id (sorted)
1.1 GB	25 MB (2%)	200 KB (0.02%)

Benchmarks

Sample data

Openstreetmap North America point data

- 1.8 billion rows
- lat/lon coordinates + 5 string columns (often null)

Format/compression	CSV / none	Parquet/Snappy	Parquet/ZSTD
Size	65 GB	32 GB	26 GB

Benchmarks

Sample data - loading

1. Convert OSM to CSV
2. Load CSV as text table in Impala
3. Rewrite table as Parquet in Impala

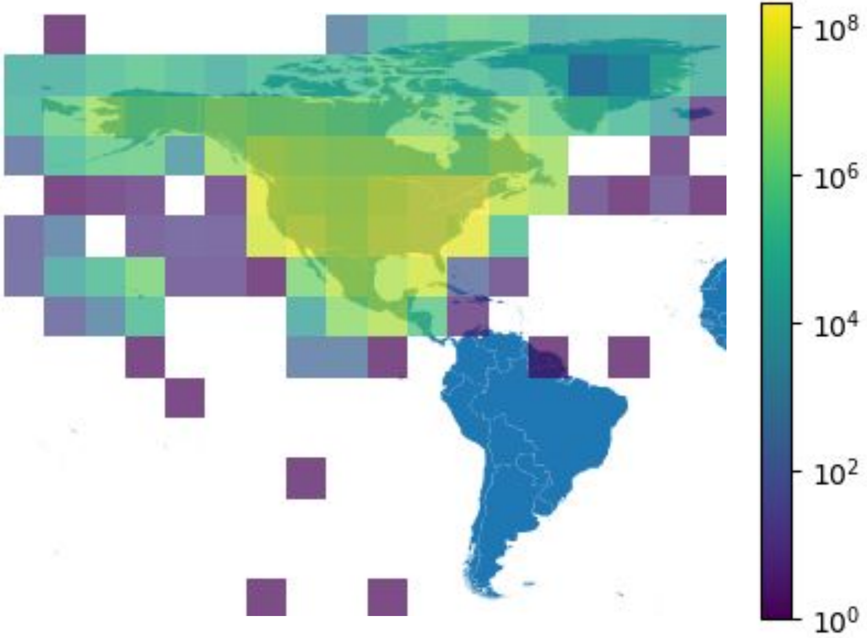
```
create table osm_north_america(
  lon DOUBLE, lat DOUBLE, cell_id BIGINT
  id STRING, name STRING, amenity STRING, shop STRING, leisure STRING
) partitioned by (bin_id bigint)
sort by (cell_id)
stored as parquet;
```

Benchmarks

Partitioning

Partition to 10° cells

- Function used: st_bin()
  - Very simple x/y cell id
  - Far from being “space filling”
- 87 non-empty partitions
- column: partition_id



Benchmarks

Partition pruning

Apply partition pruning:

```
WHERE st_intersects(
  st_binenvelope(10, partition_id),
    st_envelope(
      st_linestring(<min_lon>, <min_lat>, <max_lon>, <max_lat>)))
```

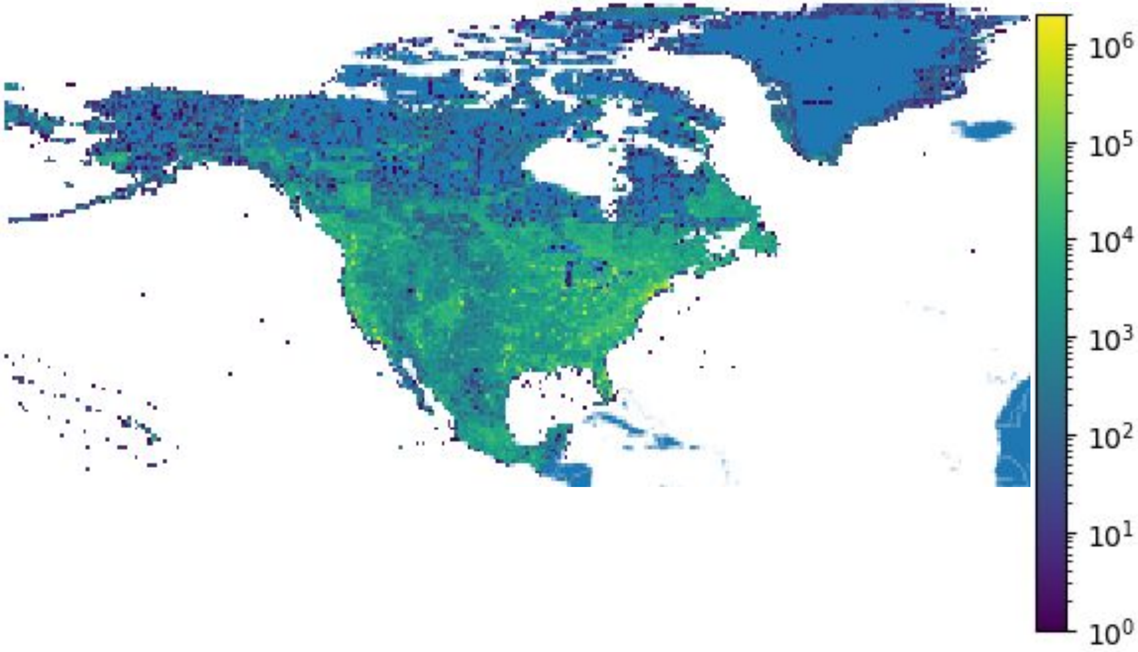
partition_id is a partitioning column -> evaluated planning time

Benchmarks

Sorting

Sort using 0.1° cell id

- Function: st_bin()
- column: cell_id



Benchmarks

Page level filtering

Using Parquet’s min-max filters need predicates on “raw” DOUBLE columns.

Bounding box filter with geospatial functions:

```
WHERE st_intersects(st_point(lon, lat), st_envelope(st_linestring(
       <min_lon>, <min_lat>, <max_lon>, <max_lat>)))
```

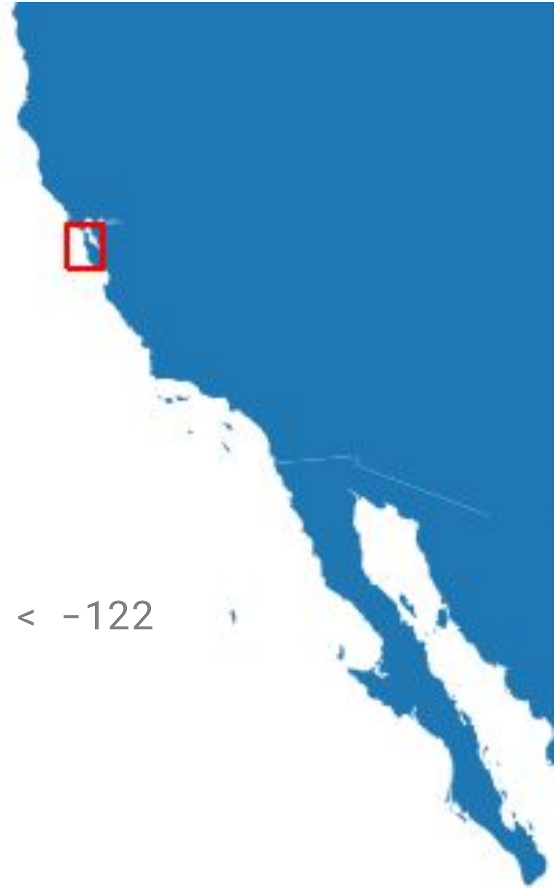
Rewrite as:

```
WHERE <min_lon> < lon AND <min_lat> < lat
  AND <max_lon> > lon AND <max_lat> < lat
```

CLOUDERA

Benchmarks

Query 1: Bay area



Rectangle around bay area

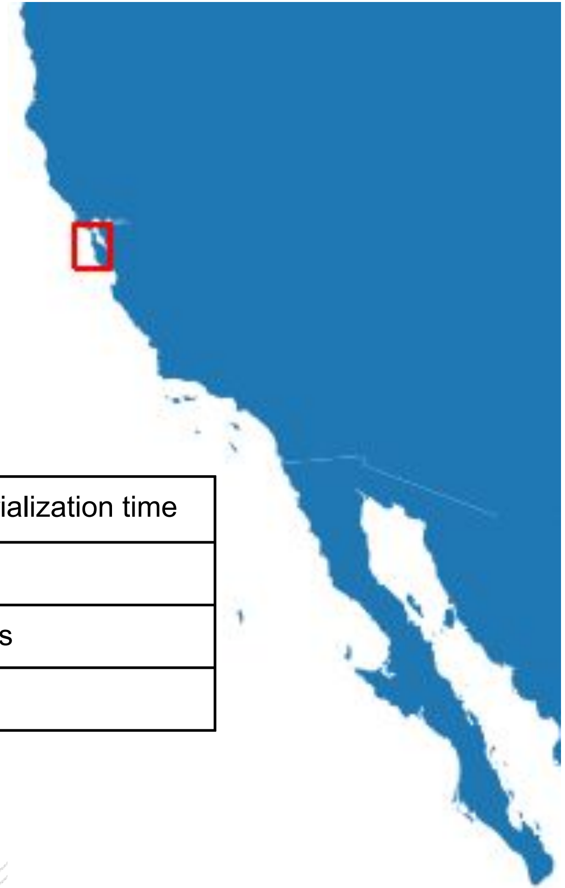
- 11M points in bounding box
- ~0.35s (single thread)*
- Dominated by decompression time (Snappy)

```
select count(*)
  from osm_north_america
  where  lat > 37 and lon > -123 and lat < 38 and lon < -122
```

*: multithreaded IO + warm cache

Benchmarks

Query 1: Bay area (details)



Rectangle around bay area

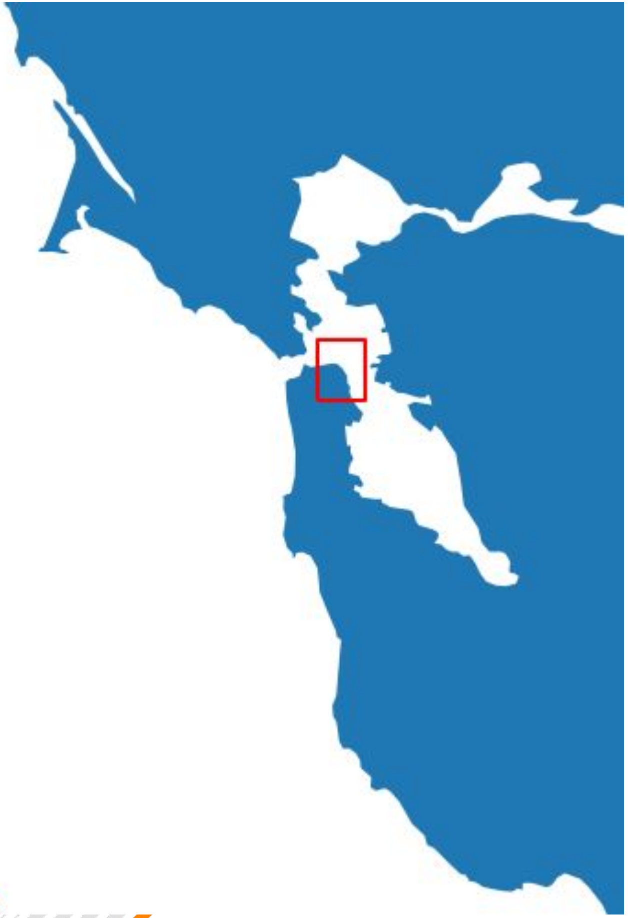
- 11M points in bounding box
- ~0.35s (single thread)*
- Dominated by decompression time

	Total time	IO bytes	IO time*	Decompression time	Materialization time
Full table scan	24s	23.3GB	6s	12s	5.2s
File level filter	0.45s	650MB	~13ms	290ms	120ms
File + page level filter	0.35s	203MB	~13ms	120ms	60ms

*: multithreaded IO + warm cache

Benchmarks

Query 2: single cell in San Francisco



Rectangle around bay area

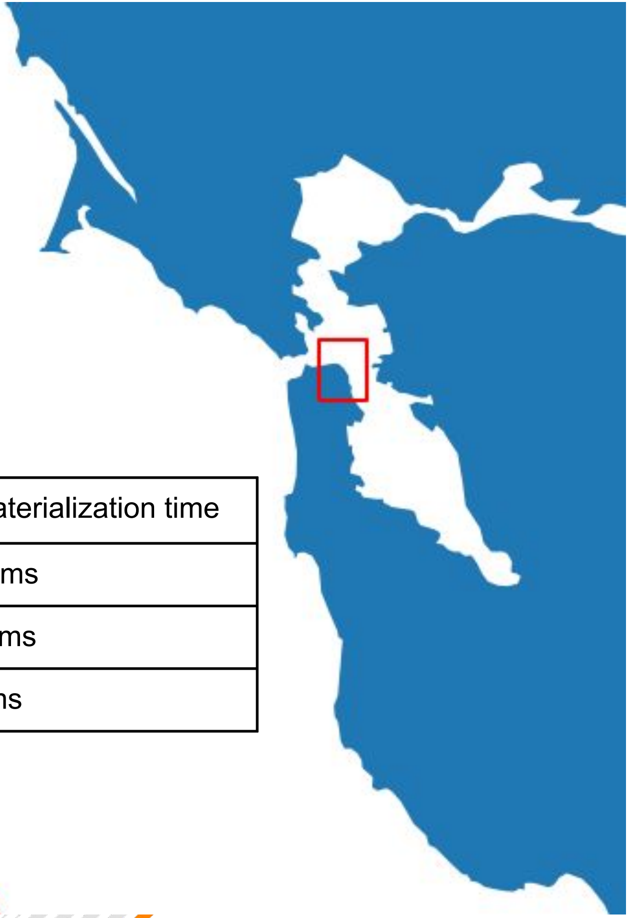
- 610K points in bounding box
- ~0.25s (single thread)*
- Dominated by decompression time (Snappy)

```
select count(*)
    from osm_north_america
    where lat > 37.75 and lon > -122.45
      and lat < 37.85 and lon < -122.35
```

*: multithreaded IO + warm cache

Benchmarks

Query 2: single cell in San Francisco



Rectangle around bay area

- 610K points in bounding box
- ~0.25s (single thread)*
- Dominated by query startup overhead

	Total time	IO bytes	IO time*	Decompression time	Materialization time
File level filter	0.45s	585MB	~12ms	280ms	86ms
File + page level filter	0.25s	12.9MB	~12ms	27ms	11ms
Cell id filter **	0.12s	1.3MB	~4ms	~0ms	4ms

*: multithreaded IO + warm cache

**: not semantically equivalent

Benchmarks

Takeaways

Min-max stats allow efficient bounding box filtering

- No code change needed, only query rewrites
- Page sizes can be reduced for more fine grained filtering

Simpler solution:

- Sort data by a space filling curve
  - allow both file and page level min/max filtering
- Use Iceberg to get planning time filtering

Benchmarks

Takeaways - decompression times

Decompression of pages can dominate execution time

- ~2x more time than bounding box check (Snappy)
- FLOAT/DOUBLE has no encoding to reduce pre-compression size

Possible improvements:

- Skip compression if not efficient
- Investigate different compressions
- Improve lazy materialization
  - Currently all predicate columns are processed eagerly

Complex geometries

- Store detailed geometry as BINARY
- Add 4 double columns to store bounding box
  - Allows min/max filtering
  - Can be large overhead from (e.g. for rectangle)
    - Bounding box can be stored at lower precision
- cell_id predicates need to handle multi-cell geometries
  - Single cell and multi-cell geometries can be separated

Questions?

Parquet with geospatial data

Point data - sorting

Sort points during insert using a space filling curve

- Groups “nearby” points together
- Pages will likely have a smaller bounding box then the whole file

Two approaches:

- “Total sorting” with z-ordering
- “Cell sorting” using cell_id from some geohash function

Parquet with geospatial data

Point data - cell_id vs z-ordering

Pros of cell sorting:

- cell_id can be used for filtering directly
- faster sorting during insert: n * log(n) -> n * log(ndv)

CLOUDERA

Parquet with geospatial data

Point data - adding cell_id column

Benchmarks use the “cell sorting” approach:

- Geohash function: st_bin(cell_size, geom) by Esri
  - Very simple, not a “real” space filling curve
- cell_id added as BIGINT column
- The goal is to have small NDV (number of distinct values) per file
  - low NDV -> very efficient encoding, minimal overhead
  - cell_id can be used for filtering directly

What file format to use for geospatial data?

File formats already supported by Impala

GeoParquet

- A project to provide a standard geospatial representation in Parquet
- Stores geometries as byte arrays in WKB format
  - probably more options will be added in the future
- Adds some JSON metadata to the Parquet header
- Pros:
  - some tools support it (e.g. GeoPandas)
- Cons:
  - would be very slow at the moment
    - needs WKB -> shapefile conversion during reading
      - Impala uses the shapefile format in memory
    - uses WKB even for point files
    - no concept of indexing (only a bounding box in the Parquet header)

What file format to use for geospatial data?

File formats already supported by Impala

HBase / Kudu

- Could be used with a geo hash included in the primary key
  - GeoMesa does something similar
- Pros:
  - efficient update/delete
  - efficient indexing
- Cons:
  - Kudu: limited scale/availability
  - HBase: inefficient range scans