How Apache Drill enables fast analytics over NoSQL databases and distributed file systems

Aman Sinha

MAPR

APACHE DRILL

About me

- Apache Drill PMC and Apache Calcite PMC. Past PMC chair of Drill.
- Employment:
  - Currently at MapR, Santa Clara. Industry's leading platform for AI and Analytics.
  - Past: ParAccel (columnar DB whose technology powers Amazon Redshift), IBM Silicon Valley Lab
- Main areas of interest: SQL query processing for RDBMS, NoSQL, Hadoop
- Contact: amansinha@apache.org, Github: https://github.com/amansinha100

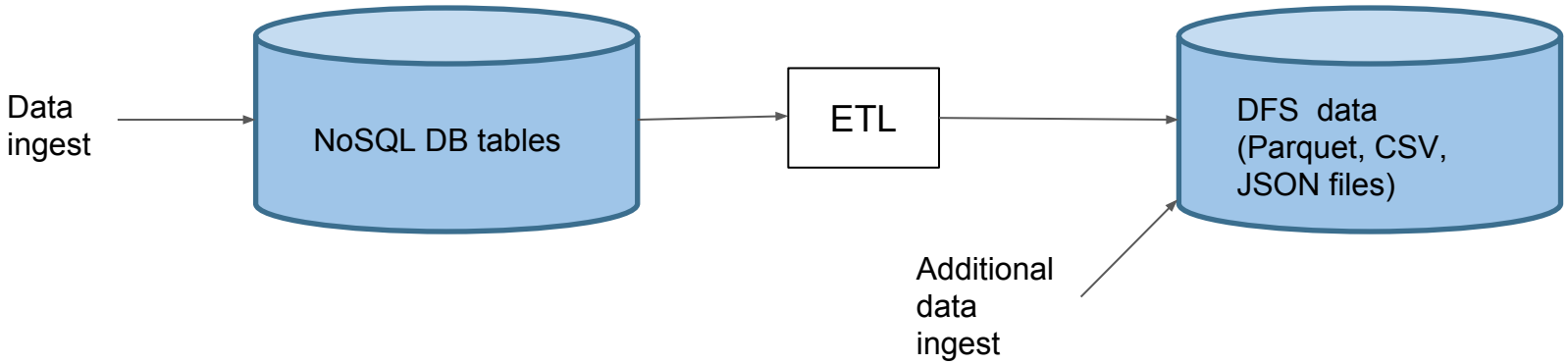
APACHE DRILL

Talk Outline

- Motivation
- Brief overview of Drill Architecture
- Improving query performance on NoSQL databases
- Improving query performance on distributed file systems
- Best practices

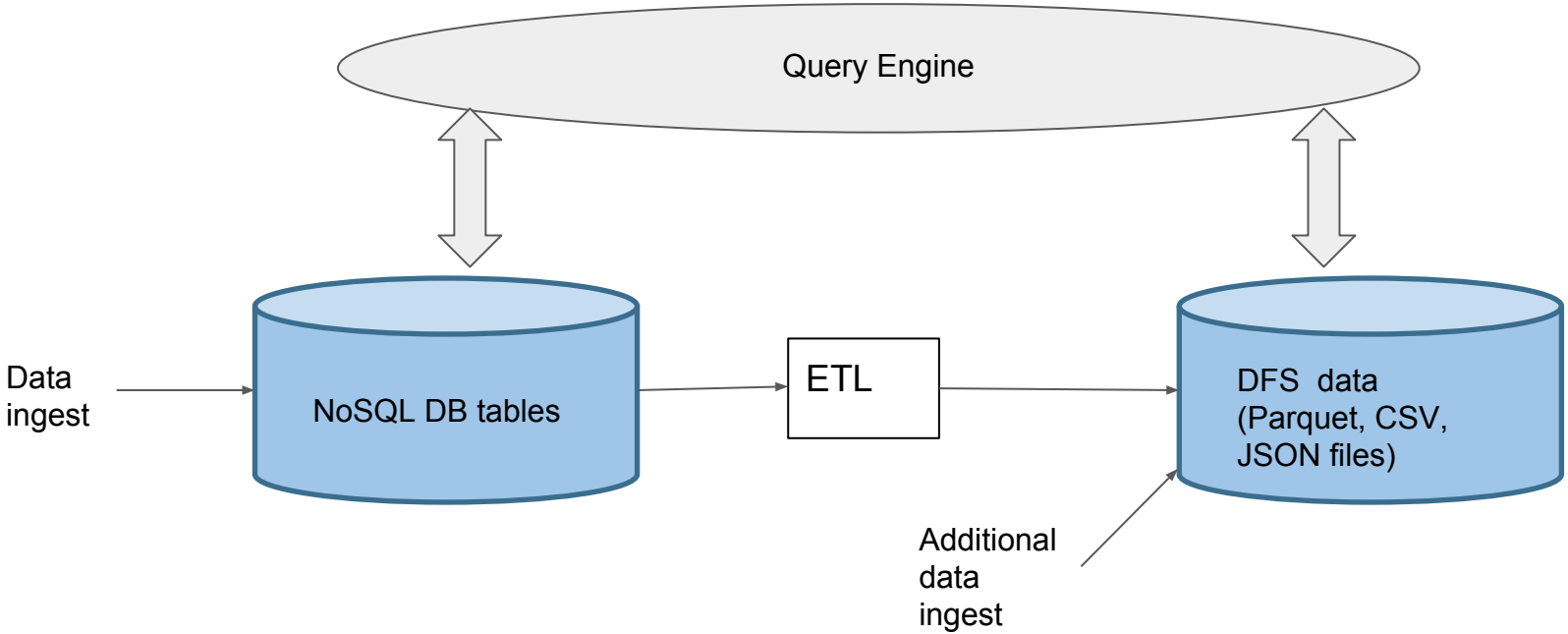
Motivation

- Enterprises often have heterogeneous data sources in the same cluster
  - Typically, NoSQL distributed databases and flat files in a distributed file system



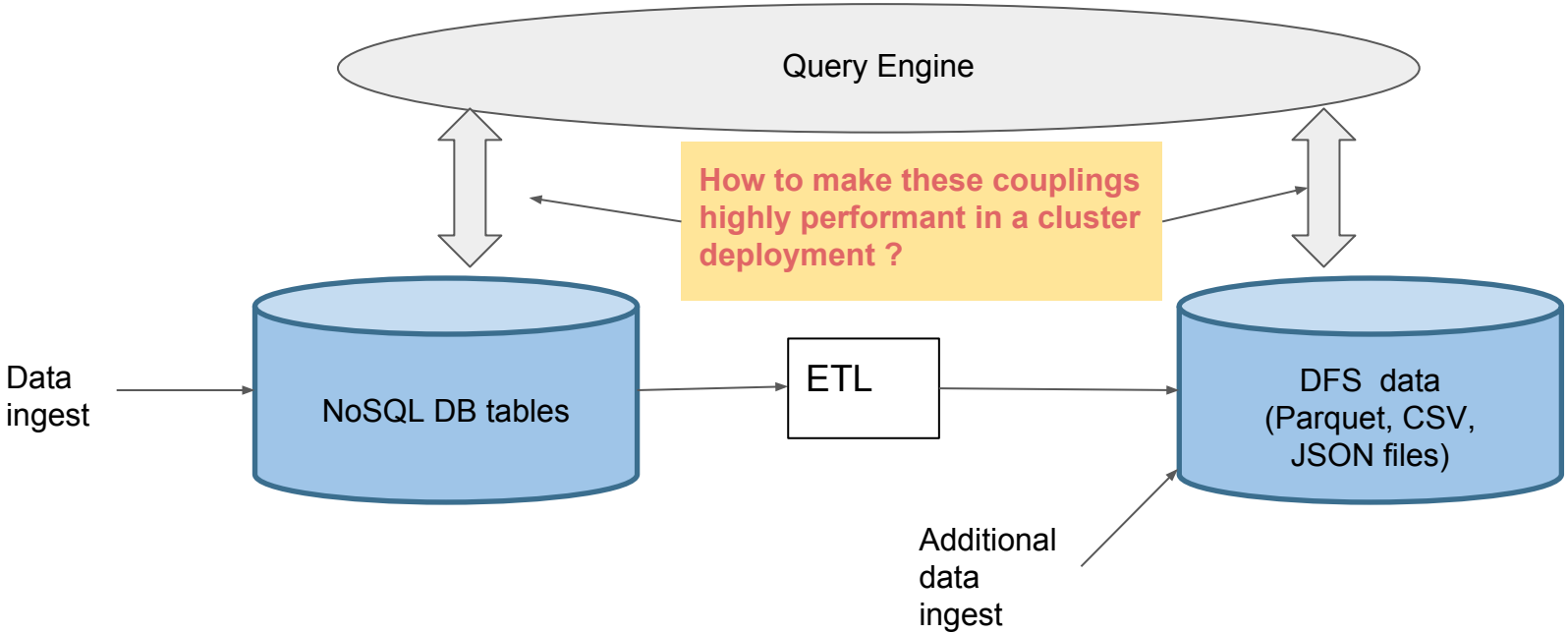
Need a Unified Query Capability

Provide SQL operations within and across data sources



APACHE DRILL

Key Challenge



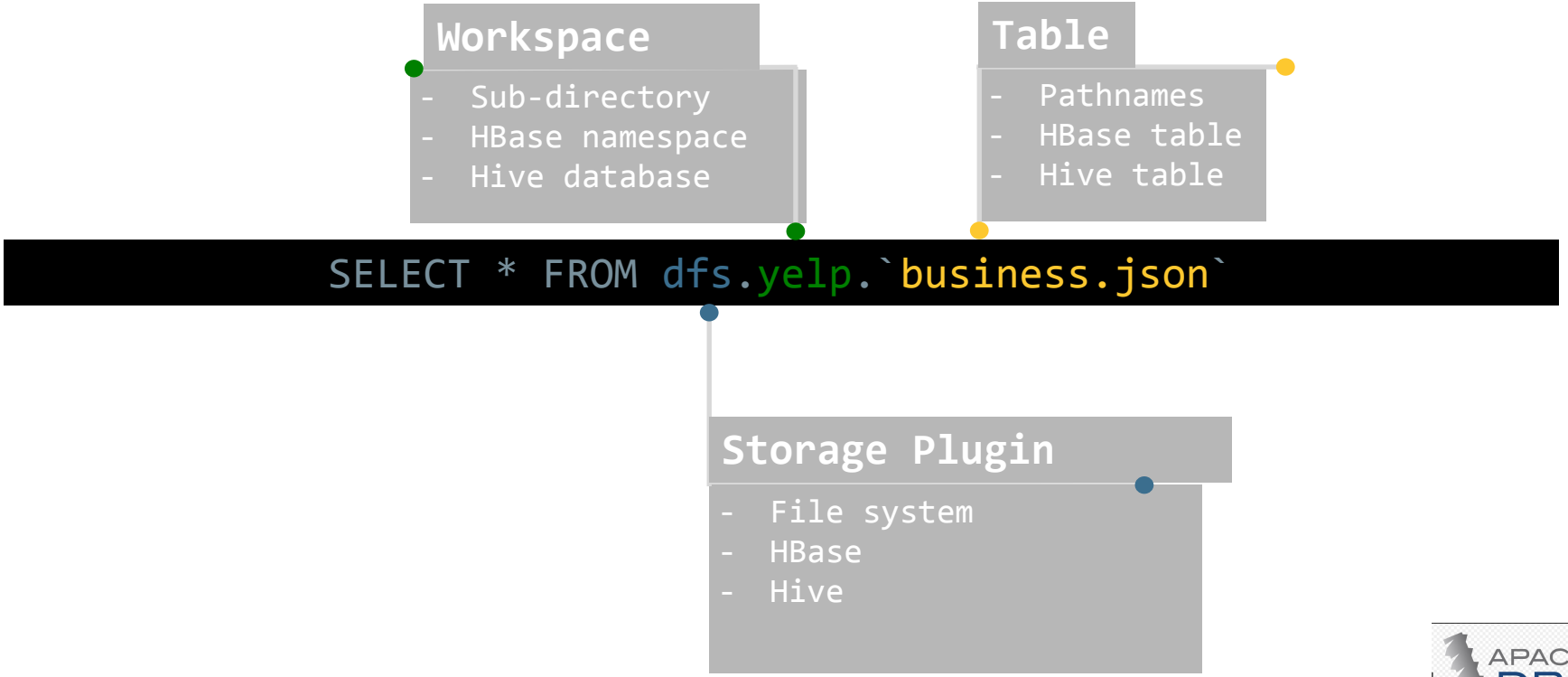
APACHE DRILL

Brief Overview of Drill Architecture

Apache Drill Introduction

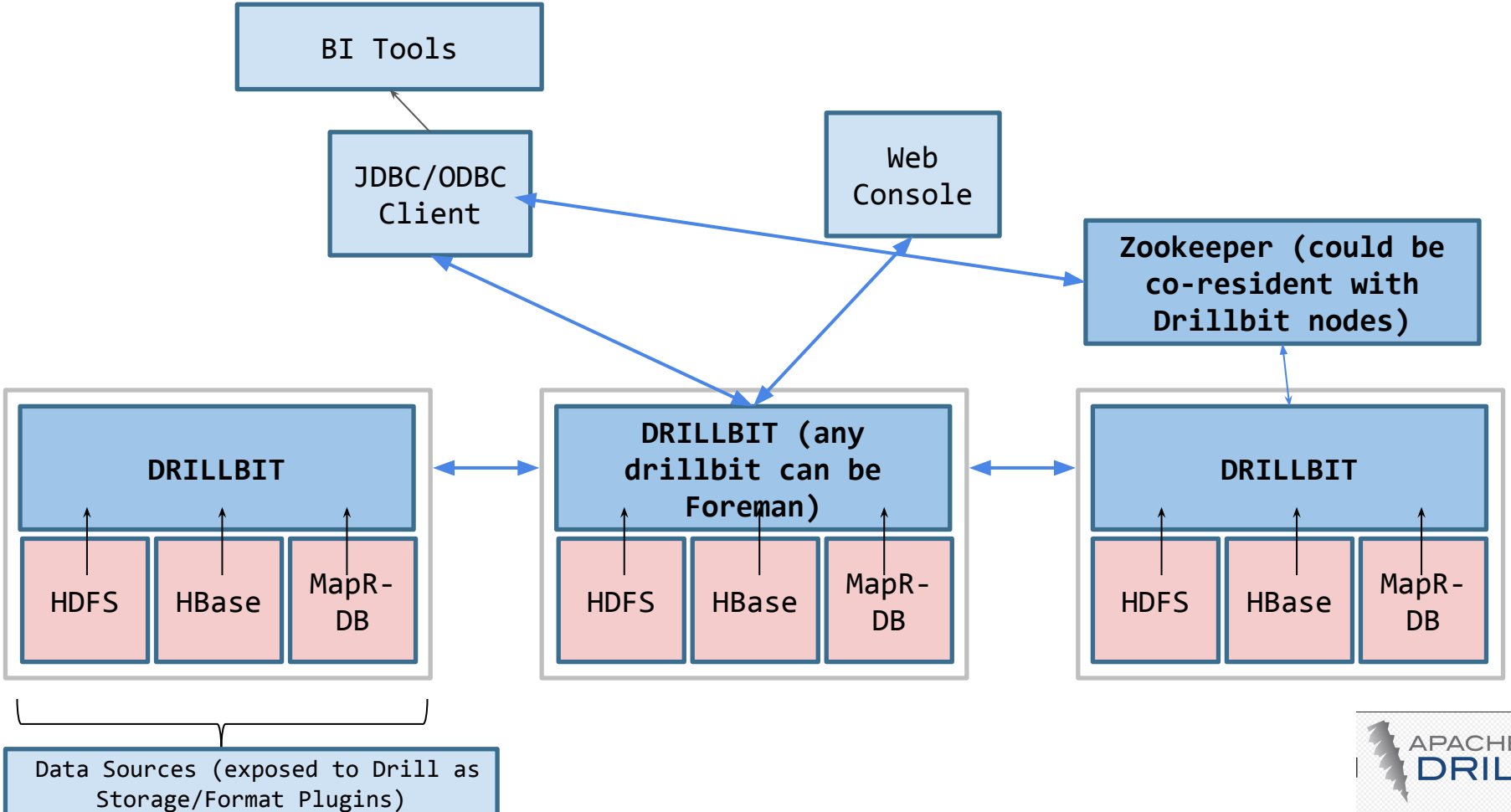
- Distributed SQL query engine
  - Originally inspired in part by Google Dremel paper
  - Became TLP in December 2014. Excerpt from the ASF announcement::
    "World's first schema-free SQL query engine brings self-service data exploration to Apache Hadoop™"
  - Supports wide range of data sources in the form of Plugins
- Data sources
  - Distributed file systems: HDFS, MapR-FS
  - NoSQL databases: HBase, MapR-DB (binary and JSON), MongoDB
  - Streaming: Kafka
  - JDBC
  - Hive (tables are managed by Hive but querying is done through Drill for interactive performance)
  - Others: OpenTSDB, Kudu, Amazon S3
- File data formats: Parquet, CSV, TSV, JSON
- Extensible

Components of a FROM Clause in Drill



APACHE DRILL

Cluster Architecture with Drill

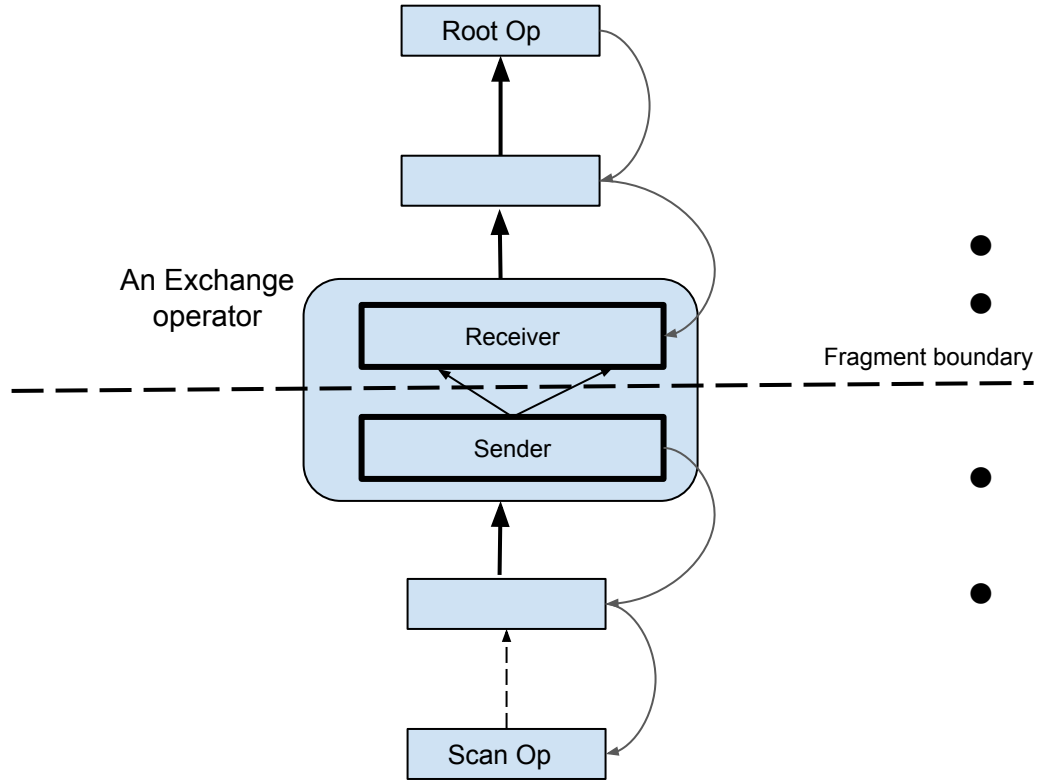


APACHE DRILL

Architecture Summary

- Schema-on-read
- No centralized metastore
- Fully Java based
- In-memory columnar processing
- Mostly off-heap memory management (negligible GC overhead)
- Code generation for run-time operators
- Optimistic, pipelined execution model
- Spill to disk for blocking operations under memory pressure
- Integrated with YARN for resource management
- Provides a strong framework for UDFs

Operator Execution Model



- Iterator based model: PULL model within a major fragment, PUSH model across major fragments
  - Several minor fragments (threads) constitute a major fragment
- Parent operator calls next() on its child
- Data is processed in 'Record Batches' (upper bounded to 64K records per batch)
- Data flow is pipelined until a blocking operator is encountered (Sort, Hash)
- Operators do run-time code generation for each new Schema

APACHE DRILL

Drill Web UI with Operator Profiles

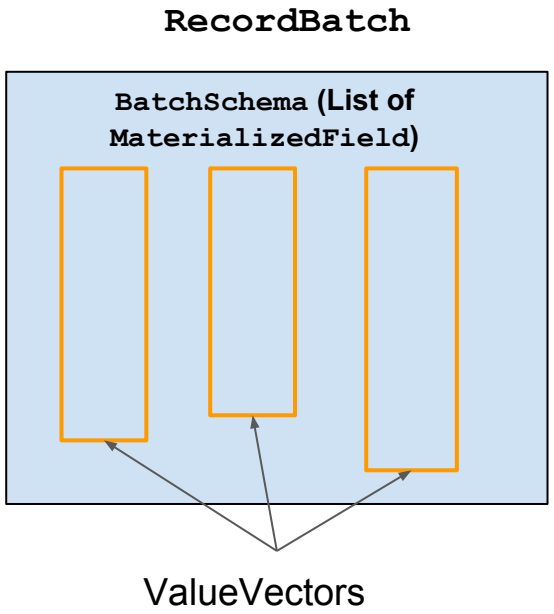
Operator Profiles

Overview

Operator ID	Type	Avg Setup Time	Max Setup Time	Avg Process Time	Max Process Time	Min Wait Time	Avg Wait Time	Max Wait Time	% Fragment Time	% Query Time	Rows	Avg Peak Memory	Max Peak Memory
00-xx-00	SCREEN	0.000s	0.000s	0.002s	0.002s	0.008s	0.008s	0.008s	8.66%	0.10%	29	-	-
00-xx-01	PROJECT	0.001s	0.001s	0.023s	0.023s	0.000s	0.000s	0.000s	83.94%	0.94%	29	-	-
00-xx-02	UNORDERED_RECEIVER	0.000s	0.000s	0.002s	0.002s	0.432s	0.432s	0.432s	7.40%	0.08%	29	-	-
01-xx-00	SINGLE_SENDER	0.000s	0.000s	0.001s	0.002s	0.005s	0.008s	0.010s	1.09%	0.18%	29	256KB	256KB
01-xx-01	HASH_AGGREGATE	0.066s	0.067s	0.126s	0.138s	0.000s	0.000s	0.000s	96.06%	15.69%	30	18MB	22MB
01-xx-02	PROJECT	0.001s	0.002s	0.003s	0.003s	0.000s	0.000s	0.000s	2.26%	0.37%	30	-	-
01-xx-03	UNORDERED_RECEIVER	0.000s	0.000s	0.001s	0.001s	0.320s	0.322s	0.323s	0.59%	0.10%	30	-	-
02-xx-00	HASH_PARTITION_SENDER	0.000s	0.000s	0.106s	0.106s	0.003s	0.003s	0.003s	97.16%	4.38%	30	64KB	64KB
02-xx-01	UNORDERED_RECEIVER	0.000s	0.000s	0.003s	0.003s	0.429s	0.429s	0.429s	2.84%	0.13%	30	-	-
03-xx-00	SINGLE_SENDER	0.000s	0.000s	0.006s	0.007s	0.014s	0.017s	0.019s	1.01%	0.79%	30	256KB	256KB
03-xx-01	PROJECT	0.031s	0.031s	0.020s	0.021s	0.000s	0.000s	0.000s	3.20%	2.50%	30	256KB	256KB
03-xx-02	HASH_AGGREGATE	0.214s	0.214s	0.523s	0.527s	0.000s	0.000s	0.000s	83.11%	64.86%	30	20MB	21MB
03-xx-03	PARQUET_ROW_GROUP_SCAN	0.000s	0.000s	0.080s	0.082s	0.014s	0.014s	0.015s	12.68%	9.90%	30	8KB	8KB

APACHE DRILL

Columnar In-memory Format



- Predecessor to Apache Arrow
- `MaterializedField` has a name and a data type
- Value Vectors are facades to byte buffers provided by underlying Netty
- Fixed-width, Variable-width value vectors
- Various data types
- 3 sub types of Value Vectors
  - Optional (Nullable)
  - Required (Non-Nullable)
  - Repeated
- Code generation: Each operator (except Scan) generates Java code at run-time based on `RecordBatch` schema

APACHE DRILL

Data Locality and Parallelization

- Drill tries to ensure that scan threads are co-located with the data file
- Encapsulated as ‘affinity’

```
List<EndpointAffinity> getOperatorAffinity();
```

Core interface methods for parallelization at the scan level

```
int getMinParallelizationWidth();

int getMaxParallelizationWidth();
```

APACHE DRILL

Back to the Performance Question …

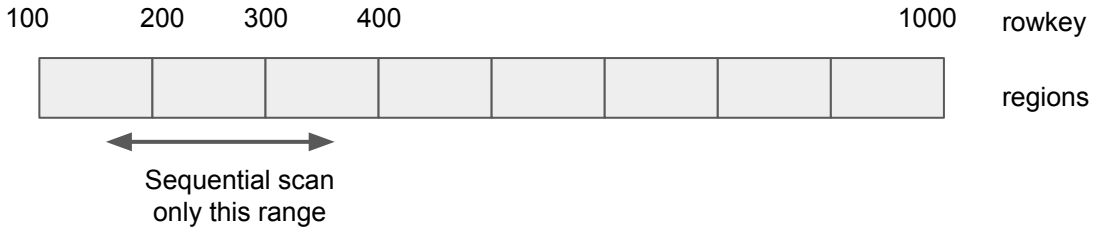
- Exploiting locality helps reduce network data transfers
- Parallelization improves CPU utilization
- **What about disk I/O ?**
  - Rest of the talk will discuss this in the context of 2 types of data sources:
    - NoSQL distributed databases
    - Distributed file systems (focus on Parquet format)

Improving query performance on NoSQL databases

Secondary Index: Background

- HBase and MapR-DB tables have primary key (row key) column
  - This column values are sorted
  - Efficient range pruning is done for rowkey predicates:

```
WHERE rowkey BETWEEN 150 AND 350
```



- Secondary columns (e.g ‘State’) values are not sorted
  - Predicate WHERE state = ‘CA’ need full table scan !
- Solution ? Create secondary index ‘tables’
  - PK of index table is a concatenation: state + rowkey :
    - **(AZ_500), … (CA_250), (CA_300), ...**

APACHE DRILL

Secondary Index

- NoSQL DBs supporting secondary index
  - MongoDB
  - MapR-DB JSON
  - HBase + Phoenix
  - Couchbase
  - Cassandra
- What's missing ?
  - Other than Hbase + Phoenix, others don't have an ANSI SQL interface
  - There's a need for a generalized cost-based index planning and execution framework
  - A key requirement:
    - Framework must be able to support 'global' non-covering indexes, not just covering index

Leveraging Secondary Index via Drill

- Storage/Format plugin whose backend supports secondary indexing
  - Reference implementation is with MapR-DB JSON
- Index metadata is exposed to Drill planner through well defined interfaces
- Statistics (if available) are also exposed
- New run-time operators added for executing index plans
- Drill Planner extends Apache Calcite's planner with additional rules for index planning.
  - Generates index-access plans and compare them cost-wise (using Apache Calcite's Volcano planner) to full table scan and with each other
- Feature will be available in upcoming Drill release (please follow JIRA: DRILL-6381, PR: https://github.com/apache/drill/pull/1466)

APACHE DRILL

Types of Queries Eligible for Index Planning

- WHERE clause with local filters
  - `<, >, =, BETWEEN`
  - `IN, LIKE`
  - Eligible ANDed conditions
  - Eligible ORed conditions
  - Certain types of functions, e.g `CAST(zipcode as BIGINT) = 12345` (only if data source supports functional indexes)
- ORDER BY
- GROUP BY (using StreamingAggregate)
- JOIN (using MergeJoin)

Leading Prefix Columns and Statistics

- Query predicate: `WHERE a > 10 AND b < 30`

Composite index key	Comments
a b c	Leading prefix columns ‘a’ and ‘b’ hence full predicate is eligible for index range pruning
a c b	Leading prefix columns only ‘a’ hence only a > 10 eligible for index range pruning

- Statistics
  - Index planning relies on statistics exposed by underlying DB (in future Drill will collect these)
  - Each individual conjunct may have an estimated row count
    - a > 10 : 2M rows
    - b < 30: 5M rows
    - Total row count of table: 100M rows

```
Selectivity = 0.02 * 0.05
```

IndexSelector uses these for cost-based analysis

APACHE DRILL

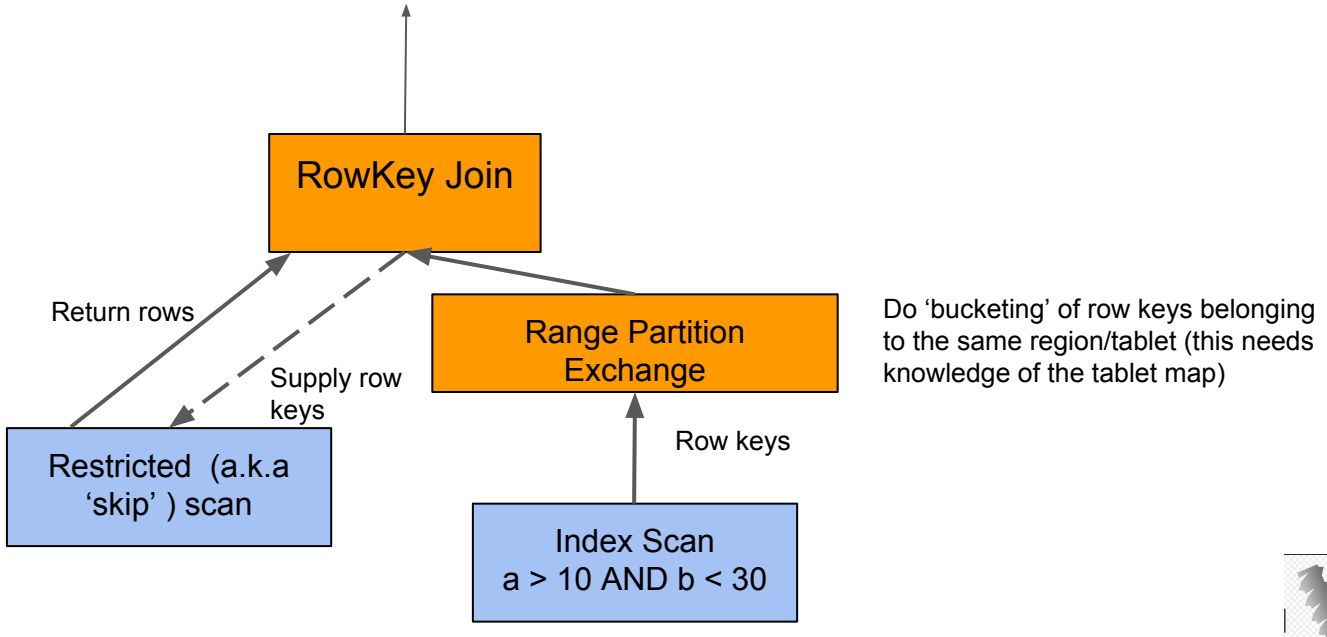
Covering vs. Non-Covering Indexes

- Covering: All columns referenced in the query are available in the index
  - Easier to handle by the planner. Generate an index-only plan.

- Non-Covering: Only a subset of the columns are available in the index
  - Needs more supporting infrastructure from planner and executor
    - RowKey join
    - Restricted (‘skip’ scan)
    - Range partitioning with a plugin-specific partitioning function

Join-back to Primary Table (for non-covering indexes)

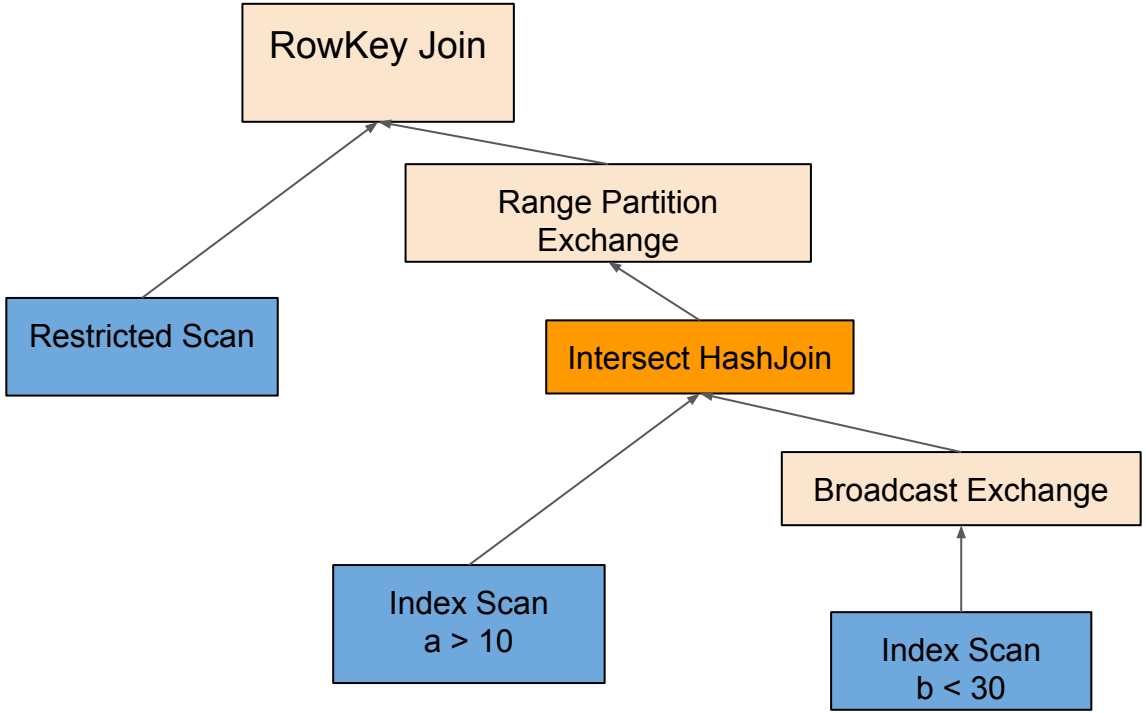
- `SELECT * FROM T WHERE a > 10 AND b < 30`
- Composite key index on {a, b}
- How to produce the remaining ('star') columns ?



APACHE DRILL

Index Intersection

- `SELECT * FROM T WHERE a > 10 AND b < 30`
- Suppose single key index exists on ‘a’ and ‘b’



APACHE DRILL

Example: GROUP BY queries

```
SELECT a, b, SUM(c)
   FROM T WHERE ..
 GROUP BY a, b
```

- Suppose composite key index exists on `{a, b}`
- Planner will create 2 types of plans
  - HashAggregate plan which does hashing on `{a, b}`
  - StreamingAggregate plan which relies on sorted input on `{a, b}`
    - The sorted input is provided by the index
    - This plan is typically cheaper than the HashAggregate plan

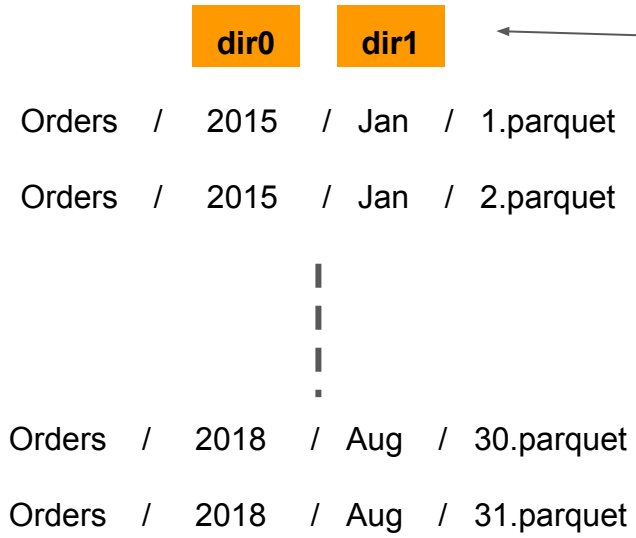
Sample interfaces to be implemented by plugin

- ***DbGroupScan***
  - `IndexCollection getSecondaryIndexCollection(RelNode scan)`
  - `DbGroupScan getRestrictedScan(List<SchemaPath> columns);`
  - `PartitionFunction getRangePartitionFunction(List<FieldReference> refList)`
  - `PluginCost getPluginCostModel()`
- ***PluginCost***
  - `int getSequentialBlockReadCost(GroupScan scan)`
  - `int getRandomBlockReadCost(GroupScan scan)`
- ***IndexDiscover***
  - `IndexCollection getTableIndex(String tableName)`
- ***IndexDefinition***
  - `List<LogicalExpression> getRowKeyColumns()`
  - `List<LogicalExpression> getIndexColumns()`
  - `List<LogicalExpression> getNonIndexColumns()`
  - `Map<LogicalExpression, RelFieldCollation> getCollationMap()`

APACHE DRILL

Improving query performance on Distributed File Systems

Directory based partition pruning



- 2 ways to query the implicit directory columns in filter conditions
  - WHERE dir0 = 2018
  - Via a view with aliasing as below

```
CREATE VIEW V1 AS
 SELECT `dir0` as `year`, `dir1` as `month`,
   o_custkey, o_totalprice, o_orderstatus
 FROM dfs.`Orders`;
```

```
SELECT o_custkey, SUM(o_totalprice)
 FROM V1
  WHERE `year` = 2018 AND
        `month` =  'July'
 GROUP BY o_custkey;
```

APACHE DRILL

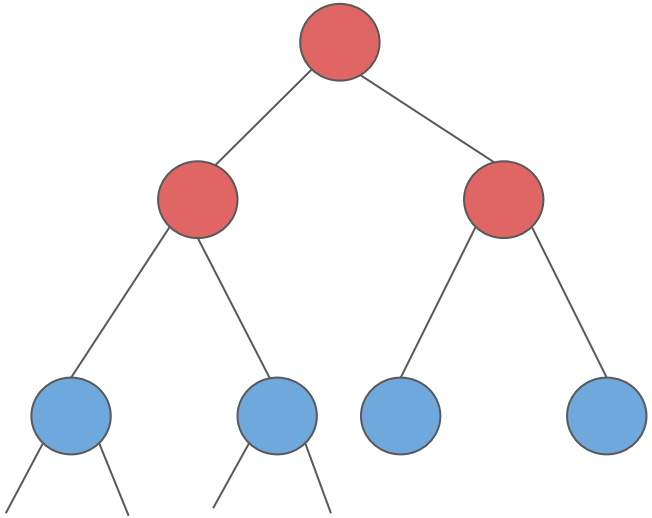
File based partition pruning (Parquet)

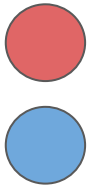
- ```
  CREATE TABLE T1 (a, b, c)
       PARTITION BY a, b
   AS SELECT col1 as a, col2 as b, col3 as c FROM …
  ```

- CTAS with Partition-By creates separate files, each file with 1 partition value
- Multiple files may be created for the same partition value
- Can prune entire file (row group) based on filter on partitioning column

APACHE DRILL
  ```

Handling Complex Predicates





Boolean operator AND, OR

Comparison operators <, >, = etc.

- The leaf nodes are either partitioning columns or non-partitioning columns or constants. E.g `` `year` = 2015 ``
- Given arbitrary expression tree, Drill will determine what predicates can be pushed down safely for partition pruning.
  - OR can only be pushed if both sides are eligible

APACHE DRILL

Parquet Row Group Metadata


```
"path" : "/Users/asinha/data/table3_meta/0_0_3.parquet",
  "length" : 245,
  "rowGroups" : [ {
    "start" : 4,
    "length" : 91,
    "rowCount" : 1,
    "hostAffinity" : {
      "localhost" : 1.0
    },
    "columns" : [ {
      "name" : "`c1`",
      "primitiveType" : "INT32",
      "originalType" : null,
      "nulls" : 0,
      "max" : 5,
      "min" : 5
    }, {
      "name" : "`c3`",
      "primitiveType" : "BINARY",
      "originalType" : "UTF8",
      "nulls" : 0,
      "max" : {
        "bytes" : "IHl5"
      },
      "min" : {
        "bytes" : "IHl5"
      }
    } ]
```


- Drill has to read RG metadata during planning time
  - Get host affinity, stats etc.
- Doesn't scale for hundreds of thousands of files
- One solution: Cache the metadata on disk by running explicit 'REFRESH TABLE METADATA <table>' command
- Can improve planning time by 10x

APACHE DRILL

Parquet Filter Pushdown

- Applied during planning process
  - Based on MIN/MAX statistics of the column in Parquet Row Group
  - Eliminates row groups after intersecting ANDed filters
- Pushdown is slightly different from partition pruning
  - Pushdown is applicable for all columns that have min/max statistics, not just partitioning columns
- Pushdown applicable for scalar columns and complex type columns
  - NOTE: Filter on VARCHAR and DECIMAL types are not currently pushed down but support is being added for upcoming Drill release

Filter Pushdown and Pruning

- **Complex Types Support**

```
SELECT * FROM table WHERE col_name.nested_col_int = 23

SELECT * FROM table WHERE col_name.nested_col_bln is not null

SELECT * FROM table WHERE col_name.nested_col_arr[0] > 10
```

- **Limitations:**
  - Logical optimizations can be applied only for the columns with available statistics

```
SELECT * FROM table WHERE col_name.nested_col_arr[2] is null
```

  - Other limitations are the same as for the Filter Pushdown for scalar data types

APACHE DRILL

Filter Pushdown and Pruning

- **Transitive Closure**

```
SELECT * FROM dfs.`/tmp/first` t1 JOIN dfs.`/tmp/SECOND` t2
ON t1.`MONTH` = t2.`MONTH` WHERE t2.`MONTH` = 4
```

Predicate pushed down on both sides of Join

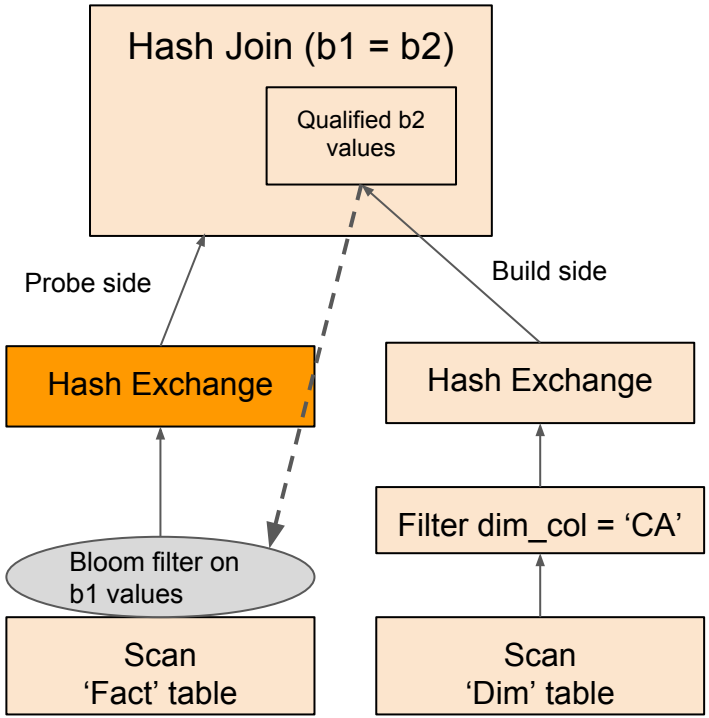
More examples:

```
SELECT * FROM t1
        JOIN t2 T2 ON t1.a = t2.a WHERE t2.a NOT IN (4, 6)
SELECT * FROM t1
        JOIN (SELECT a, b FROM table WHERE  a = 1987 AND b = 5) t2
          ON t1.a = t2.a AND t1.b = t2.b
```

APACHE DRILL

Run-Time Filter Pushdown for HashJoin

```
SELECT a1 FROM fact WHERE b1 IN (SELECT b2 FROM dim WHERE dim_col = 'CA')
```



Goal: Substantially reduce network transfer by this exchange

APACHE DRILL

Best Practices for Parquet Data Layout

- Large number of small files or fewer large files ?
  - Recommended to have fewer large files subject to constraints such as Parquet block size and desired parallelism
    - Too many small files (order of few KBs) may cause data skew and affect performance
    - Too few large files reduces parallelism … need to determine sweet spot
- Block size considerations
  - Best to have Parquet block size equal or less than HDFS/MapR-FS block size
    - Larger block size will spread 1 row group over 2 or more nodes incurring remote reads.
- Preferred compression type
  - Snappy is preferred for performance, gzip for space but higher cost to decompress
  - Default for Drill - snappy

Summary

- Apache Drill's extensible architecture makes it easy to support multiple data sources
  - While exploiting data locality and parallelism suited to the data source
- Fast analytics on NoSQL databases using a new generalized framework to do index based planning and execution
- Fast analytics on distributed file system tables by doing intelligent partitioning and filter pushdown

APACHE DRILL

Useful links

- Drill resources
  - http://drill.apache.org
  - Twitter: @ApacheDrill
  - Mailing lists:
    - user@drill.apache.org
    - dev@drill.apache.org
    -

Get involved with the Drill community !

Q & A