cloudera

Nested Types in Impala

Alex Behm // Cloudera, Inc.

Marcel Kornacker // Cloudera, Inc.

Skye Wanderman-Milne // Cloudera, Inc.

Impala Meetup 03/24/2015

Design Goals

- **Goals**
  - Support for nested data types: struct, map, array
  - Full expressiveness of SQL with nested structures
- **v1 Prioritization**
  - Focus on SELECT queries (INSERT in later releases)
  - Focus on native Parquet and Avro formats (XML, JSON, etc in later releases)
  - Focus on built-in language expressiveness (UDTF extensibility in later releases)

Example Schema

```
CREATE TABLE Customers {
  id BIGINT,
  address STRUCT<
            city: STRING,
            zip: INT
          >
  orders ARRAY<STRUCT<
           txn_time: TIMESTAMP,
           cc: BIGINT,
           items: ARRAY<STRUCT<
                    item_no: STRING,
                    price: DECIMAL(9,2)
                  >>
         >>
  preferred_cc BIGINT
}
```

cloudera

Impala Syntax Extensions

- Path expressions extend column references to scalars (nested structs)
- Can appear anywhere a conventional column reference is used

Find the ids and city of customers who live in the zip code 94305:

SELECT id, **address.city** FROM customers WHERE **address.zip** = 94305

- Collections (maps and arrays) are exposed like sub-tables
- Use FROM clause to specify which collections to read like conventional tables
- Can use JOIN conditions to express join relationship (default is INNER JOIN)

Find all orders that were paid for with a customer's preferred credit card:

SELECT o.txn_id FROM customers c, **c.orders o** WHERE o.cc = c.preferred_cc

cloudera

Referencing Arrays & Maps

- Basic idea: Flatten nested collections referenced in the FROM clause
- Can be thought of as an implicit join on the parent/child relationship

```
SELECT c.id, o.txn_id FROM customers c, c.orders o
```

id of a customer repeated for every order

c.id	o.txn_id
100	*203*
100	*305*
100	*507*
…	…
101	*10056*
101	*10*
…	…

cloudera

Referencing Arrays & Maps

```
SELECT c.id, o.txn_id FROM customers c, c.orders o
SELECT c.id, o.txn_id FROM customer c INNER JOIN c.orders o
```

Returns customer/order data for customers that have at least one order

```
SELECT c.id, o.txn_id FROM customers c LEFT OUTER JOIN c.orders o
```

Also returns customers with no orders (with order fields NULL)

```
SELECT c.id, o.txn_id FROM customers c LEFT ANTI JOIN c.orders o
```

Find all customers that have no orders

cloudera

Motivation for Advanced Querying Capabilities

- Count the number of orders per customer

```
SELECT COUNT(*) FROM customers c, c.orders o GROUP BY c.id
```

Must be unique

- Count the number of items per order

```
SELECT COUNT(*) FROM customers.orders o, o.items GROUP BY ???
```

- **Impractical → Requires unique id at every nesting level**
- Information is already expressed in nesting relationship!

- What about even more interesting queries?
- Get the number of orders and the average item price per customer?
  - "Group by" multiple nesting levels

cloudera

Advanced Querying: Correlated Table References

- Count the number of orders per customer

```
SELECT cnt FROM customers c, (SELECT COUNT(*) cnt FROM c.orders) v
```

Correlated reference to “c”

- Count the number of items per order

```
SELECT cnt FROM customers.orders o, (SELECT COUNT(*) cnt FROM o.items) v
```

- Get the number of orders and the average item price per customer

```
SELECT c.id, cnt, avgp
FROM customer c,
  (SELECT count(1) cnt FROM c.orders) v1,
  (SELECT avg(price) avpg FROM c.orders.items) v2
```

cloudera

Advanced Querying: Correlated Table References

- **Full expressibility**
  - Arbitrary SQL allowed in inline views and subqueries with correlated table refs
  - Correlated subqueries transformed into joins if possible

```
SELECT id FROM customers c WHERE EXISTS
  (SELECT 1 FROM c.orders o, o.items i where o.cc = c.preferred_cc and i.price > 100)
```

- **Exploits nesting relationship**
  - No need for stored unique ids at various levels
- Similar to standard SQL ‘LATERAL’, ‘CROSS APPLY’, ‘OUTER CROSS APPLY’
  - Goes beyond standard in some aspects (semi/anti variants)
  - Not as general as SQL standard (limited correlations)

cloudera

Impala Execution Model

Design sweet spot: small/medium sized collections (max few hundreds of MB)

- Built-in limitation on the size of nested collections (TBD)
- Huge collections not expected to be performant and rare

Execution Overview

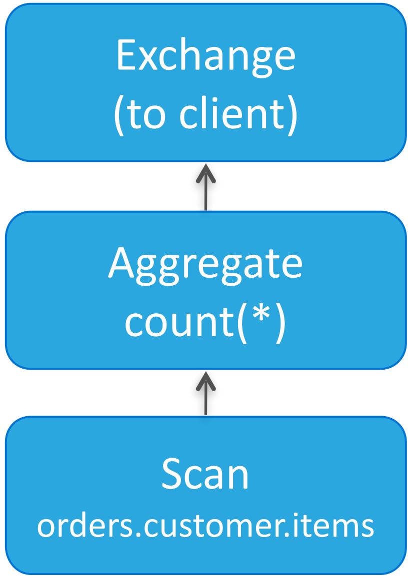
- Scans materialize minimal nested structure in memory
- Three new exec nodes
  - **Subplan**: Executes its subplan tree for each input row and returns the rows produced by the subplan
  - **Nested Row Src:** Returns the current input row of its parent Subplan node
  - **Unnest**: Scans an array slot of the current input row of its parent Subplan node or of an output row from its child plan node, returning one row per array element.

cloudera

Example A

Count the total number of items

```
SELECT count(*) FROM customer.orders.items
```



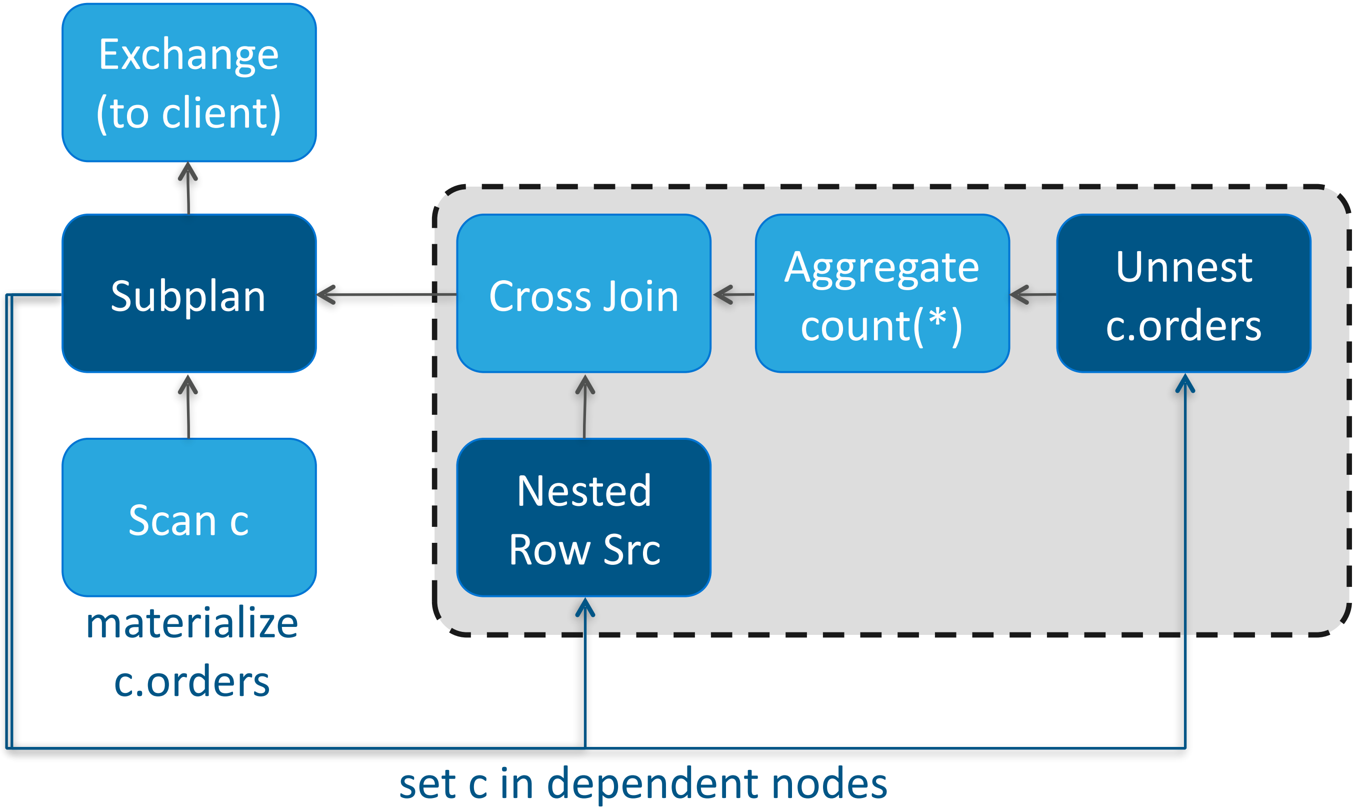
Example B

Count the number of orders per customer

```
SELECT c.id, cnt
FROM customer c,
JOIN (SELECT count(*) cnt FROM c.orders) v
```

Subplan Pseudo-Code

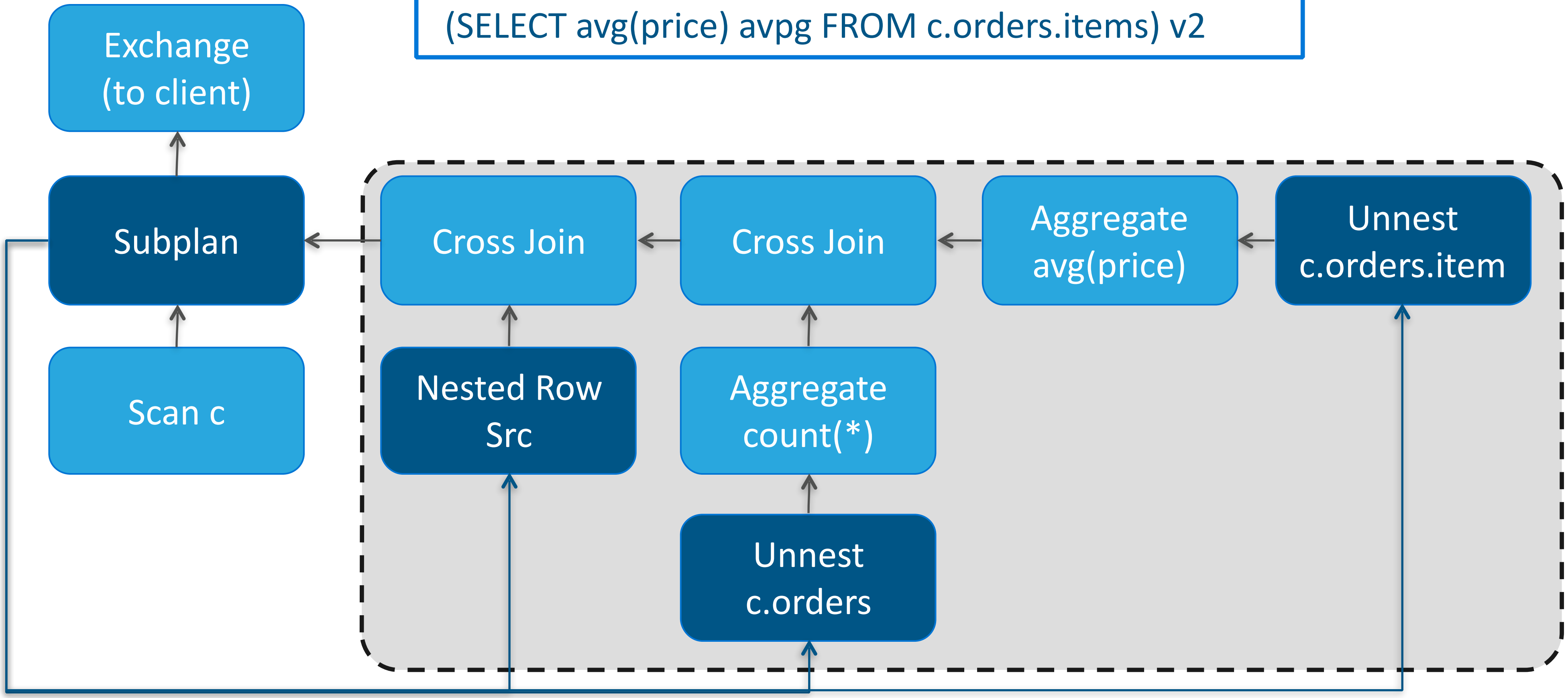
```
result = {}
for each c in input
  set c in dependent nodes
  result += rhsPlan.exec()
return result
```



Example C

Return the number of orders and
the average item price per customer

```
SELECT c.id, cnt, avgp
FROM customer c,
  (SELECT count(*) cnt FROM c.orders) v1
  (SELECT avg(price) avpg FROM c.orders.items) v2
```

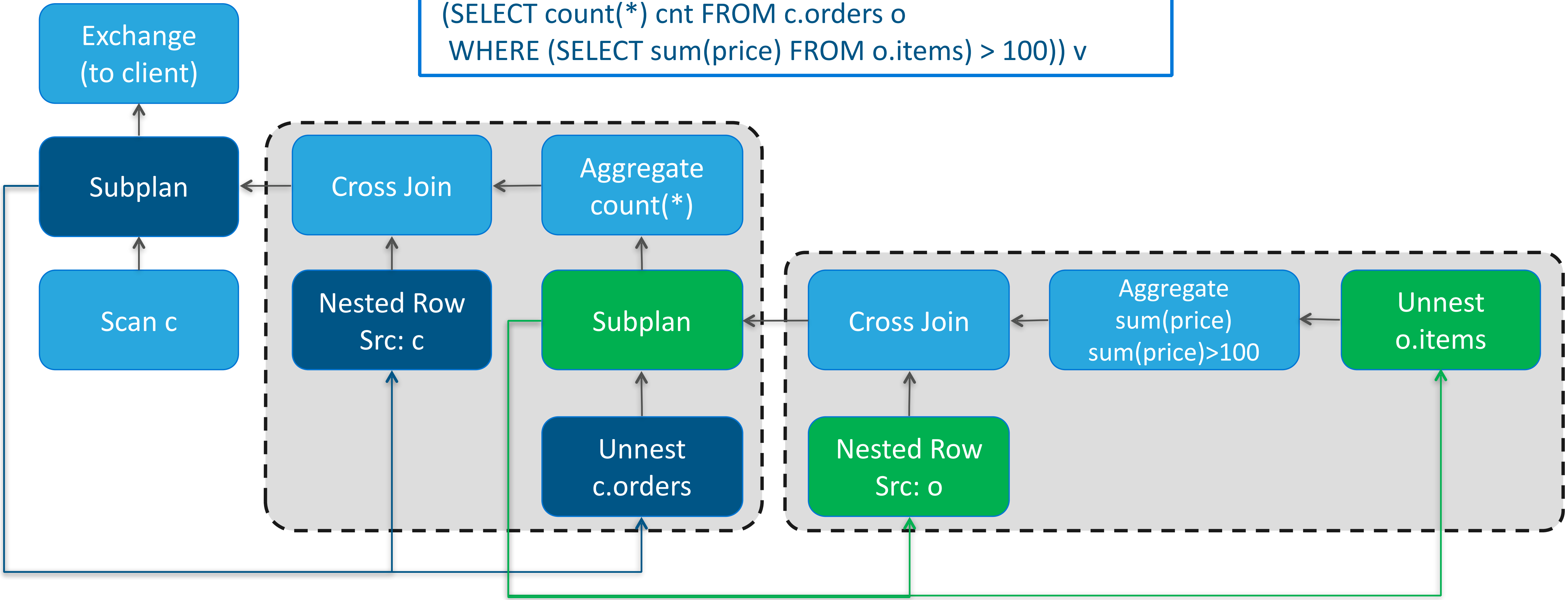


cloudera

Example D

For each customer, return the number of orders
whose total item price exceeds > 100

```
SELECT c.id, cnt, avgp
FROM customer c,
  (SELECT count(*) cnt FROM c.orders o
   WHERE (SELECT sum(price) FROM o.items) > 100)) v
```



cloudera

Future Work

- Syntax extensions

```
SELECT    c.id,
          count(c.orders),
          avg(c.orders.items.price)
FROM customer c
```

Rewrite

```
SELECT c.id, cnt, avgp
FROM customer c,
  (SELECT count(1) cnt FROM c.orders) v1
  (SELECT avg(price) avpg FROM c.orders.items) v2
```

- Performance improvements
  - Parquet: Scan columns directly in “unnest”, avoid collection materialization
  - Codegen subplans
- INSERT queries, e.g., convert flat data into nested data
- UDTF support
- More formats: JSON, XML, etc.

cloudera

Thank you

Appendix

Comparison of Impala and HiveQL

- **Impala's syntax provides a superset of Hive's functionality**

- **HiveQL has similar path expressions but with restrictions**
  - Must use LATERAL VIEW in FROM clause; more verbose syntax
  - LATERAL VIEWs themselves have many restrictions, no arbitrary SQL
  - Requires complex joins or unique ids at various nesting levels for expressing even simple queries (e.g., find number of orders per customer)
  - Does not provide similar inline view and subquery capabilities

cloudera

Impala Syntax vs. Hive Syntax

Impala	Hive
SELECT … FROM customer c, c.orders o, o.items i	SELECT … FROM customer c LATERAL VIEW explode(c.orders) as (c1, c2…) LATERAL VIEW explode(c.orders.items) as (c1, c2…)
SELECT … FROM customer c LEFT JOIN c.orders LEFT JOIN c.orders.items	SELECT … FROM customer c OUTER LATERAL VIEW explode(c.orders) as (c1, …) OUTER LATERAL VIEW explode(c.orders.items) as (c1, …)
SELECT c.id FROM customer c WHERE NOT EXISTS (SELECT oid FROM c.orders WHERE c.preferred_cc = orders.cc) SELECT c.id FROM customer c, LEFT ANTI JOIN (SELECT oid FROM c.orders) v ON c.preferred_cc = v.cc	No convenient/performant equivalent in Hive.

cloudera

Impala Syntax vs. Hive Syntax

```
SELECT count(c.orders) FROM customer c
```

```
SELECT cnt FROM customer c,
JOIN (SELECT count(1) cnt FROM c.orders) v1
```

```
SELECT count(1)
FROM customer c
LATERAL VIEW explode(c.orders) as (c1, c2…)
GROUP BY c.orders
(in absence of a unique key in ‘customer’)
```

Impala will not support Hive’s builtin or user-defined table generating functions for now.

```
SELECT …
FROM customer c
LATERAL VIEW MY_UDTF(c.orders) as (c1, c2…)
```

```
SELECT …
FROM customer c
LATERAL VIEW json_tuple(c.json_str, …) as (c1, c2…)
```

cloudera

Impala Syntax vs. Hive Syntax

```
SELECT    c.id,
          count(c.orders),
          avg(c.orders.items.price)
FROM customer c
```

```
SELECT c.id, cnt, avgp
FROM customer c,
JOIN (SELECT count(1) cnt FROM c.orders) v1
JOIN (SELECT avg(price) avpg FROM c.orders.items) v2
```

No convenient/performant equivalent in Hive.

- Impala is more expressive, but less extensible until UDTFs are supported
  - More join types: inner/outer/semi/anti
  - Full SQL block inside correlated inline view
- Hive more extensible (UDTFs), has builtin UDTFs
- Hive Lateral View very rigid, no arbitrary SQL inside

Impala Nested Types in Action

Josh Will's Blog post on analyzing misspelled queries

http://blog.cloudera.com/blog/2014/08/how-to-count-events-like-a-data-scientist/

- **Goal:** Rudimentary spell checker based on counting query/click events
- **Problem:** Cross referencing items in multiple nested collections
  - Representative of many machine learning tasks (Josh tells me)
- Goal was not naturally achievable with Hive
- Josh implemented a custom Hive extension "WITHIN"
- **How can Impala serve this use case?**

Impala Nested Types in Action

```
SELECT a.qw, a.qr, count(*) as cnt
FROM sessions
LATERAL VIEW WITHIN(
  "SELECT bad.query qw, good.query qr
  FROM t1 as bad, t1 as good
  WHERE bad.tstamp_sec < good.tstamp_sec
  AND good.tstamp_sec - bad.tstamp_sec < 30
  AND bad.event_id NOT IN (select search_event_id FROM t2)
  AND good.event_id IN (select search_event_id FROM t2)",
search_events, install_events) a
GROUP BY a.qw, a.qr;
```

Josh’s HiveQL extension

```
SELECT bad.query, good.query, count(*) as cnt
FROM sessions s,
  s.search_events bad,
  s.search_events good,
WHERE bad.tstamp_sec < good.tstamp_sec
  AND good.tstamp_sec - bad.tstamp_sec < 30
  AND bad.event_id NOT IN (select search_event_id FROM s.install_events)
  AND good.event_id IN (select search_event_id FROM s.install_events),
GROUP BY bad.query, good.query;
```

Impala SQL

Schema

```
account_id: bigint
search_events: array<struct<
  event_id: bigint
  query: string
  tstamp_sec: bigint
  ...
>>
install_events: array<struct<
  event_id: bigint
  search_event_id: bigint
  app_id: bigint
  ...
>>
```

cloudera

Impala Nested Types in Action

```
SELECT bad.query, good.query, count(*) as cnt
FROM sessions s,
  s.search_events bad,
  s.search_events good,
WHERE bad.tstamp_sec < good.tstamp_sec
  AND good.tstamp_sec - bad.tstamp_sec < 30
  AND bad.event_id NOT IN (select search_event_id FROM s.install_events)
  AND good.event_id IN (select search_event_id FROM s.install_events),
GROUP BY bad.query, good.query;
```

Rewrite

```
SELECT bad.query, good.query, count(*) as cnt
FROM sessions s
JOIN (SELECT * FROM s.search_events) bad,
JOIN (SELECT * FROM s.search_events) good,
LEFT ANTI JOIN (SELECT search_event_id FROM s.install_events) v1
ON (bad.event_id = v1.install_events)
LEFT SEMI JOIN (SELECT search_event_id FROM s.install_events) v2
ON (good.event_id = v2.search_event_id)
WHERE bad.tstamp_sec < good.tstamp_sec
  AND good.tstamp_sec - bad.tstamp_sec < 30
GROUP BY bad.query, good.query;
```

cloudera

Impala Nested Types in Action

```
SELECT bad.query, good.query, count(*) as cnt
FROM sessions s
JOIN (SELECT * FROM s.search_events) bad,
JOIN (SELECT * FROM s.search_events) good,
LEFT ANTI JOIN (SELECT search_event_id FROM s.install_events) v1
ON (bad.event_id = v1.install_events)
LEFT SEMI JOIN (SELECT search_event_id FROM s.install_events) v2
ON (good.event_id = v2.search_event_id)
WHERE bad.tstamp_sec < good.tstamp_sec
  AND good.tstamp_sec - bad.tstamp_sec < 30
GROUP BY bad.query, good.query;
```