Impala Intro

MingLi
xunzhang
Overview

- MPP SQL Query Engine for Hadoop Environment
- Designed for great performance
- BI Connected (ODBC/JDBC, Kerberos, LDAP, ANSI SQL)
- Hadoop Components
  - HDFS, HBase, Metastore, Yarn, Sentry…
  - Parquet, Avro, RCFile, …
- Runs on same nodes running Hadoop processes
History

- Developed by Cloudera

- Versions
  - Released as beta in 10/2012
  - 1.0 version available in 05/2013
  - Current version 2.1

- Open Source
  - github.com/cloudera/impala
  - github.com/apache/incubator-impala
Architecture

- Impalad: daemon process
- Each node handling requests simultaneously
- Metadata management
  - catalog service (DDL, third-party stores, external metadata)
    - single node
- System state repository and distribution
  - statestore (pub-sub proxy)
  - single node
Impalad

- Frontend (Java)
  - parse queries
  - plan queries
- Backend (C++)
  - coordinate
  - execute plan fragments
- Local cache of metadata
- RPC/Comm
  - thrift
Query Execution

- Query execution phases
  - request via ODBC/JDBC
  - create plan fragments
  - coordinator initiates execution

- During execution
  - Intermediate results are streamed between executors
  - query results are streamed back to client
  - blocking operators
    - top-n, aggregation, sorting
Planning

* 2-phase process
  * I. single-node planning
  * II. plan parallelization and fragmentation
Single-Node plan

```
SELECT t1.custid, SUM(t2.revenue) AS revenue
FROM t1
JOIN t2 ON (t1.id1 = t2.id)
JOIN t3 ON (t1.id2 = t3.id)
WHERE t3.category = 'Online'
GROUP BY t1.custid
ORDER BY revenue DESC LIMIT 10;
```
Distributed Plan

**Goals**
- max scan locality
- min data movement
- add exchange plan node
- add non-exchange plan node

**Parallel join**
- broadcast join: large x small
- partitioned join: large x large

**Split out Fragments**
- Cost based decision based on column stats/estimated cost of data transfer
Execution Engine

- Implemented in C++
- Leverages decades DB research
  - partitioned parallelism
  - pipelined relational operators
  - batch at a time runtime
- Focussed on speed and efficiency
  - intrinsics/machine code for text parsing, hashing, etc
  - runtime code generation with LLVM
Runtime Code-gen

- JIT compile
- Effect as custom-coding query
  - remove branches, unroll loops
  - propagate constants, offsets, pointers, etc
  - inline function calls (virtual function)
- Optimized for modern CPUs (pipeline)
1.3x for all the TPC-H queries combined!
I/O Management

- Fixed number of worker threads
  - one I/O thread per rotational disk
  - eight I/O thread per SSD
- Different operators with different method
  - join: single thread
  - aggregation: single thread
  - read threads + scan threads
- HDFS short-circuit local reads from DN
- HDFS centralized caching for small FACT table (no copy data blocks?)
Case Study 1

/* TPC_H Query 1 - Pricing Summary Report */

SELECT L_RETURNFLAG, L_LINESTATUS, SUM(L_QUANTITY) AS SUM_QTY,
    SUM(L_EXTENDEDPRICE) AS SUM_BASE_PRICE, SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)) AS SUM_DISC_PRICE,
    SUM(L_EXTENDEDPRICE*(1-L_DISCOUNT)*(1+L_TAX)) AS SUM_CHARGE, AVG(L_QUANTITY) AS AVG_QTY,
    AVG(L_EXTENDEDPRICE) AS AVG_PRICE, AVG(L_DISCOUNT) AS AVG_DISC, COUNT(*) AS COUNT_ORDER
FROM LINEITEM
WHERE L_SHIPDATE <= dateadd(dd, -90, cast('1998-12-01' as date))
GROUP BY L_RETURNFLAG, L_LINESTATUS
ORDER BY L_RETURNFLAG, L_LINESTATUS

* Fragment 2
   * scan LINEITEM
   * apply predicate
   * perform partial aggregation
   * shuffle across all the nodes

* Fragment 1
   * merge partial aggregates(generate 1 row of data in 4 nodes)

* Fragment 0
   * merge all partial aggregates
   * sort results
Case Study I

Table 3: Time Breakdown for Query 1 in Hive-MR

<table>
<thead>
<tr>
<th>Phase</th>
<th>TXT (secs)</th>
<th>ORC (secs)</th>
<th>ORC Snappy (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR1-map phase</td>
<td>485</td>
<td>215</td>
<td>176</td>
</tr>
<tr>
<td>MR1-reduce phase</td>
<td>5</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>MR2</td>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 4: Time Breakdown for Query 1 in Impala

<table>
<thead>
<tr>
<th>Format</th>
<th>TXT (secs)</th>
<th>Parquet (secs)</th>
<th>Parquet Snappy (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>F2</td>
<td>71</td>
<td>22</td>
<td>21</td>
</tr>
</tbody>
</table>

- launch mappers and reducers
- CPU-bound (deserialization)
- disk utilisation <20%
- column format reduce deser amount
- multi-threads: read threads + scan threads
- Parquet (196GB) | ORC (223GB)
- code-gen for aggregation
- CPU-bound
- disk utilization ~ 85%
Case Study II

/* TPC_H Query 17 - Small-Quantity-Order Revenue */

SELECT SUM(L_EXTENDEDPRICE)/7.0 AS AVG_YEARLY FROM LINEITEM, PART
WHERE P_PARTKEY = L_PARTKEY AND P_BRAND = 'Brand#23' AND P_CONTAINER = 'MED BOX'
AND L_QUANTITY < (SELECT 0.2*AVG(L_QUANTITY) FROM LINEITEM WHERE L_PARTKEY = P_PARTKEY)
Case Study II

- correlation optimization
- scan twice
- single-thread aggregation
### Table 1: Hive vs. Impala Execution Time

<table>
<thead>
<tr>
<th>TPC-H Query</th>
<th>Hive - MR</th>
<th>Hive - Tez</th>
<th>Impala</th>
<th>ORC-Snappy (Hive-Tez) over Parquet-Snappy (Impala)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>TXT (secs)</td>
<td>ORC (secs)</td>
<td>ORC Snappy (secs)</td>
<td>TXT (secs)</td>
</tr>
<tr>
<td>Q1</td>
<td>535</td>
<td>266</td>
<td>228</td>
<td>232</td>
</tr>
<tr>
<td>Q2</td>
<td>293</td>
<td>299</td>
<td>294</td>
<td>130</td>
</tr>
<tr>
<td>Q3</td>
<td>761</td>
<td>543</td>
<td>462</td>
<td>524</td>
</tr>
<tr>
<td>Q4</td>
<td>534</td>
<td>277</td>
<td>232</td>
<td>272</td>
</tr>
<tr>
<td>Q5</td>
<td>1078</td>
<td>833</td>
<td>772</td>
<td>831</td>
</tr>
<tr>
<td>Q6</td>
<td>357</td>
<td>109</td>
<td>85</td>
<td>114</td>
</tr>
<tr>
<td>Q7</td>
<td>1409</td>
<td>1096</td>
<td>1038</td>
<td>581</td>
</tr>
<tr>
<td>Q8</td>
<td>1085</td>
<td>822</td>
<td>770</td>
<td>743</td>
</tr>
<tr>
<td>Q9</td>
<td>3371</td>
<td>2883</td>
<td>2848</td>
<td>2102</td>
</tr>
<tr>
<td>Q10</td>
<td>778</td>
<td>526</td>
<td>502</td>
<td>336</td>
</tr>
<tr>
<td>Q11</td>
<td>235</td>
<td>234</td>
<td>216</td>
<td>158</td>
</tr>
<tr>
<td>Q12</td>
<td>555</td>
<td>308</td>
<td>241</td>
<td>226</td>
</tr>
<tr>
<td>Q13</td>
<td>255</td>
<td>268</td>
<td>228</td>
<td>176</td>
</tr>
<tr>
<td>Q14</td>
<td>417</td>
<td>175</td>
<td>155</td>
<td>124</td>
</tr>
<tr>
<td>Q15</td>
<td>619</td>
<td>272</td>
<td>233</td>
<td>166</td>
</tr>
<tr>
<td>Q16</td>
<td>247</td>
<td>235</td>
<td>239</td>
<td>152</td>
</tr>
<tr>
<td>Q17</td>
<td>1041</td>
<td>779</td>
<td>813</td>
<td>1205</td>
</tr>
<tr>
<td>Q18</td>
<td>1183</td>
<td>925</td>
<td>859</td>
<td>1128</td>
</tr>
<tr>
<td>Q19</td>
<td>907</td>
<td>677</td>
<td>656</td>
<td>919</td>
</tr>
<tr>
<td>Q20</td>
<td>663</td>
<td>379</td>
<td>332</td>
<td>236</td>
</tr>
<tr>
<td>Q21</td>
<td>2230</td>
<td>1605</td>
<td>1495</td>
<td>1688</td>
</tr>
<tr>
<td>Q22</td>
<td>363</td>
<td>385</td>
<td>452</td>
<td>123</td>
</tr>
<tr>
<td>AM</td>
<td>860</td>
<td>632</td>
<td>598</td>
<td>553</td>
</tr>
<tr>
<td>GM</td>
<td>665</td>
<td>459</td>
<td>422</td>
<td>358</td>
</tr>
<tr>
<td>AM-Q{9,21}</td>
<td>666</td>
<td>470</td>
<td>440</td>
<td>419</td>
</tr>
<tr>
<td>GM-Q{9,21}</td>
<td>577</td>
<td>393</td>
<td>360</td>
<td>303</td>
</tr>
</tbody>
</table>
Roadmap

- More SQL
  - ROLLUP/GROUPING SETS
  - INTERSECT/MINUS
  - MERGE
- Nested data structures
  - structs, arrays, maps in Parquet, Avro, Json
- Additional data types: DATA, TIME, DATETIME
- Multithread execution past scan
- Automated Data Conversion
- Support S3-backed tables, SAN-based Systems
- Improved query planning (more elaborate statistics, histograms)
Potential Pros

- MPP
- Parquet format data skipping
- Code generate
- I/O bound queries with great performance
- Shared-nothing database architecture
- Adjust resource consumption estimates during execution, accumulating resource allocation
Potential Cons

- MPP
- No ORC (lightweight index)
- Executors bundled
- Can not recover from mid-query failure
- Single thread execution of joins and aggregations
  - CPU bound queries
  - complex queries
HAWQ vs Impala

- Multi-threads vs Virtual Segments
- Shared-nothing vs Master-slave
  - multiply user performance
- Libhdfs3 vs Libhdfs(JNI)
- Resource Management Strategy(TBD)
Related Discussion

- [HAWQ-303] - Index support for non-heap tables
- [HAWQ-312] - Multiple active master support
- [HAWQ-319] - REST API for HAWQ
- [HAWQ-786] - Framework to support pluggable formats and file systems
- [HAWQ-864] - Support ORC as a native file format
Reference


SQL-on- Hadoop: Full circle back to shared-nothing database architectures. VLDB14.
The Truth

https://www.youtube.com/watch?v=tHxZsM8N0HQ