Bonusproject 2
Exam Style
RR Debugger
Execution Engines

Timo Kersten
Technische Universität München
Faculty for Computer Science
Chair for Database Systems
Return Trips

Task: Identify possible return trips.

```
select *
from tripsProvided a, tripsProvided b
where
distance(a.droplocation, b.pickuplocation) < r and
distance(b.droplocation, a.pickuplocation) < r and
a.droptime < b.pickuptime and
a.droptime + 8 hours > b.pickuptime
```

Yay, spark sql can take this as input!

Oh no, the optimiser gets very confused, resorts to using a cross product.
Bucketize

Oh no, the optimiser gets very confused, resorts to using the cross product.

Let’s help it out by introducing an additional equijoin on a grid.

... 

explode(...-1,...,+1) // per dimension

...

to.as("to").join(back.as("back"),
   "$to.dropTimeBucket" === "$back.pickupTimeBucket" &&
   "$to.latBucketDropoff" === "$back.latBucketPickup" &&
   "$to.lonBucketDropoff" === "$back.lonBucketPickup" &&
   "$to.tpep_dropoff_datetime" < "$back.tpep_pickup_datetime" &&
   "$back.tpep_pickup_datetime" < "$to.tpep_dropoff_datetime" + lit(dt * 60 * 60) &&
makeDistExpr("$back.dropoff_latitude", "$back.dropoff_longitude",
   "$to.pickup_latitude", "$to.pickup_longitude") < lit(dist) &&
makeDistExpr("$to.dropoff_latitude", "$to.dropoff_longitude",
   "$back.pickup_latitude", "$back.pickup_longitude") < lit(dist)
)
Bucket Computation

Bucket width in degrees for latitude

\[ \frac{\Delta \text{lat}}{360} = \frac{\text{dist}}{\text{earthRadius} \times 2\pi} \]

Bucket width in degrees for longitude

\[ \frac{\Delta \text{lon}}{360} = \frac{\text{dist}}{\text{smallCircleRadius} \times 2\pi} \]

\[ \text{smallCircleRadius} = 2\pi \times \text{earthRadius} \times \sin(90^\circ - \text{maxLat}) \]
Trade Offs

More dimensions for bucketing
+ Increases selectivity of bucketization
- Increases dataset size by factor of 3 per dimension

Few dimensions for bucketing
+ Smaller intermediate result
- Many elements per bucket, large crossproducts within buckets
Bottleneck During Join
Partitioning Writes to Disk
Further Optimizations

We want to use all dimensions for selectivity, but only use one dimension to reduce communication.

Idea: Map all dimensions onto a space-filling curve
Z-order curve

- Space filling curve
- \((x,y)\) to z-curve computed by interleaving bits
- We could map our bucket numbers from all dimensions to z-curve,
- only explode for left neighbor, self, and right neighbor once
Other optimization: Bloom filter

- Bloom filters are very small set representations that allow for membership tests
- False positives possible, no false negatives

We can use it after building partitions for one side of the join:
- Partition one side
- Build bloom filters for partitions
- Broadcast bloom filters
- Check bloom filters before sending
- Drop element if result negative
- -> reduce amount of tuples in shuffle phase
Last Year’s Leaderboard

- ThisTeam: What is going on here?
- Other tips:
  - Overall bottleneck: Shuffle phase writes to disk
  - Strike a balance between selective bucketing and too much data created by explode/union
  - Don’t use udfs, these need to deserialize data to jvm objects -> garbage collection

<table>
<thead>
<tr>
<th>#</th>
<th>Team</th>
<th>Runtime(s)</th>
<th>Badges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ThisTeam</td>
<td>0.703</td>
<td>🏆🚗conds</td>
</tr>
<tr>
<td>2</td>
<td>ross</td>
<td>73.756</td>
<td>🏆🚗conds</td>
</tr>
<tr>
<td>3</td>
<td>Sherlock</td>
<td>101.541</td>
<td>🏆🚗conds</td>
</tr>
<tr>
<td>4</td>
<td>PNC</td>
<td>112.205</td>
<td>🏆🚗conds</td>
</tr>
<tr>
<td>5</td>
<td>tessa</td>
<td>115.174</td>
<td>🏆🚗conds</td>
</tr>
</tbody>
</table>

Bonus Task: Taxi Rides
Thinking Outside the Box (a.k.a a Hack)

Task:

Put your implementation into ReturnTrips.scala and assure that ReturnTrips.compute(tripsProvided, dist, sparkContext) returns a dataset with all trips and their return trips. That means, each row in the returned dataset must contain a trip and a return trip, so that all trips in tripsProvided are returned with their return trips in case they have any.

Idea:

dist only varies from 50 to 200m. Calculate the join before time measurement with list = 200m. Filter down on actual request:

```scala
prejoined.filter(
    makeDistExpr("back.dropoff_latitude", "back.dropoff_longitude",
        "to.pickup_latitude", "to.pickup_longitude") < lit(dist) &&
    makeDistExpr("to.dropoff_latitude", "to.dropoff_longitude",
        "back.pickup_latitude", "back.pickup_longitude") < lit(dist)
)
```

Another suggestion:

dist only has 150 possible values. Precompute result for all of them.
Exam Style

- 90 Minutes, 90 Points -> ~ 1 point per minute
- Tasks have different difficulty levels, so it may be a good idea to skip ahead if you are stuck

- Pay attention to what the questions are asking for
  - Name
  - Name and give and example
  - Name and explain
  - State a SQL query that finds all...
  - Is it possible that …? Explain why or why not.

- Don’t write long stories. Answer the questions concisely to get all points but save time for other questions.
Topics Covered

GNU tools (grep etc.)
Performance Spectrum/Estimations
Machine-code optimizations
Advanced SQL:
  - Recursive SQL
  - Query Decorrelation
  - Window Functions
Distributed Databases (2PL, Partitioning, Replication)
Map-Reduce (Map, Shuffle, Reduce, Exploit Parallelism!)
Scale up vs. Scale out
No-SQL Databases
Distributed Hash-Tables
XML, JSON, RDF, SPARQL
... and more (this is not the definitive list)
RR Debugger

Programs run backwards in time
Repeatable race conditions

rr-project.org
Usage:
Run program with ‘rr record <program cmd line>’
Debug with ‘rr replay’

Variable inspection and breakpoints as usual in gdb
+ reverse-next, reverse-continue

Interesting workflow:
Why am I reading a nullptr from this memory location?
-> Set hardware watchpoint to memory location
-> reverse-continue
Stops debugger when memory location was last written

Alternatives to RR
Chronon for Java
RevDeBug for .Net/C#
RevPDB for Python
UndoDB for compiled code
Time Travel Debugger in Visual Studio Enterprise