Search Space Improvements: Cost-Based Logical Transformations

This document discusses the state-of-the-art concerning cost-based application of logical and physical transformations, and Presto’s current limitations in this area. The discussion also presents detailed design considerations for removing these limitations. This document is one in a series of search space-related discussion documents that focus on expanding and better controlling the space of plan alternatives that the Presto optimizer considers.

In the sections that follow we cover

1. Examples of logical transformations that should be considered within a cost-based evaluation model
2. The state of current cost-based transformations in Presto
3. A proposal to implement the Cascades optimizer framework in Presto, with a brief description of how logical and physical transformations would be explored, costed and memoized
4. Key design considerations and open questions to be considered during implementation

In the Appendix, we also consider existing implementations of the Cascades frameworks in some other popular relational databases

Background

Relational query languages provide a high-level declarative interface for accessing relational data. The job of the query optimizer is to translate a declarative query into an execution plan that correctly and efficiently returns the final query result [1]. The optimization process involves enumerating alternative execution plans from a search space of feasible alternatives and choosing one that minimizes a cost estimate of the efficiency of the execution plan. The optimizer generates its search space by applying transformation rules that logically map a query to equivalent algebraic representations and implementation rules that map those algebraic representations to execution plans consisting of physical operators, executable by the query evaluation engine.

Examples of cost-based logical plan transformations

Group By / Distinct view merging

Consider the below query (uses TPCH tables):

```
--Q1

SELECT o1.orderdate,
    l.receiptdate
FROM orders o1,
    lineitem l,
     (SELECT orderkey, AVG (o2.totalprice) as avgTotalPrice
        FROM orders o2
      GROUP BY orderkey) o2
WHERE o1.orderkey = l.orderkey
    and o1.orderkey = o2.orderkey
    and l.extendedprice > 104948
    and o1.totalprice > avgTotalPrice
```

We first compute the AVG(totalprice) for *all* orders (In the o2 subquery), and then join the results with the output of `o1 IJ l`

This query can also be written as:

```
--Q1 GROUP BY pull up

SELECT V.orderdate,
    V.receiptdate
FROM (
        select
            l.receiptdate,
            o1.orderdate,
            o1.totalprice,
            avg(o2.totalprice) OVER (PARTITION BY o2.orderkey ROWS BETWEEN UNBOUNDED
PRECEDING AND UNBOUNDED FOLLOWING) as avgPricePerOrderKey
        FROM orders o2,
            orders o1,
            lineitem l
        WHERE o1.orderkey = l.orderkey
            and l.extendedprice > 104948
            and o2.orderkey = o1.orderkey
    ) V
WHERE V.totalprice > V.avgPricePerOrderKey
```

Here we are effectively performing the `GROUP BY` *after* the JOIN. This makes sense cost-wise if the filter `l.extendedprice > 104948` is effectively able to reduce the orders data that needs to be aggregated

Early out joins

Transforming existential queries to early out joins can simplify a query from one that uses a SEMIJOIN to an INNERJOIN, thereby opening up the JOIN space-

```
SELECT <cols/expressions>

FROM a

WHERE <expression1> IN (SELECT <expression2> FROM b)

Can be converted to -

SELECT DISTINCT id,

sq1.< cols / expressions >

FROM (

SELECT uuid() AS id,

< cols / expressions >,

<expression1>

FROM a

) sq1 INNER JOIN

(

SELECT <expression2>

FROM b

) sq2 ON sq1.<expression1> = sq2.<expression2>;
```

NNER JOIN's between relations can be reordered more easily (see ReorderJoins), so it is possible that we get a cheaper plan than the first one which uses a Semi Join

Query Decorrelation

Consider the Q1 query again. This query is a decorrelated version of the below correlated query -

```
--Q1 Correlated

SELECT o1.orderdate,
    l.receiptdate
FROM orders o1,
    lineitem l
WHERE o1.orderkey = l.orderkey
    and l.extendedprice > 104948
    and o1.totalprice > (
        SELECT AVG (o2.totalprice)
        FROM orders o2
        WHERE o2.orderkey = o1.orderkey
    )
```

The predicate: `l.extendedprice > 104948` reduces the cardinality of the `o1 InnerJoin l` by 99.99%, thereby making the case that a (hypothetical) `ApplyNode` over the Subquery block would have lower execution time and should be of cheaper cost

Predicate pushdown/pullup

Presto currently pushes down predicates as far as possible to the TableScan nodes, assuming that these would reduce the amount of data processed earlier in the operator pipeline. This is a good heuristic, but not always true. Consider the plans for the two logically equivalent queries -

```
-- Q1: All predicates pushed down
select l.quantity from lineitem l, orders o where l.orderkey = o.orderkey and
o.totalprice < 1000 and l.quantity < 48;

-- Q2: ` l.quantity + floor(random())` evaluated after the JOIN
select l.quantity from lineitem l, orders o where l.orderkey = o.orderkey and
o.totalprice < 1000 and l.quantity + floor(random()) < 48;
```

The `floor(random())` expression that is added to l.quantity in Q2 always evaluates to `0` so the effective predicate of `l.quantity < 48` remains the same in both queries. However, the filter is pushed down to the TableScan node (see appendix) in Q1, but is evaluated *after* the cardinality reducing INNER JOIN in Q2

For the TPCH SF1 schema, Presto assigns the below CPU cost estimate to the queries -

`Q1: 290,640,426`

`Q2: 210,051,887`

Indeed, we observe that Q2 is how lower latency and uses lesser CPU at runtime (see gist).
This is because the `l.quantity` has a distinct value range of `[1,50]`, and the predicate `l.quantity < 48`, when evaluated on the lineitem table, only reduces the row count by about 5%. The JOIN however reduces cardinality by 99.97 %. By delaying the evaluation of the predicate to be after the JOIN, we avoid evaluating the predicate on rows that would not qualify for the JOIN anyhow.

Additionally, the cost of predicate evaluation is assumed to be equal in Presto - so all filters are pushed down without considering their relative cost. However, not all predicates have equal cost. E.g. :
`startswith(sha256(quantity),xxxx)` should be costed more than `quantity < 48`

Presto Design Considerations

This section discusses the current state of the Presto optimizer with respect to application of logical transformations. It also discusses design considerations for extending these capabilities by incorporating the previously discussed state-of-the-art for making the application of certain types of logical transformations cost-based.

Current state of Presto cost-based optimizations

Presto currently does cost based optimizations only in a few iterative optimizer rules: ReorderJoins (and related Join rules[4]), PushPartialAggregationThroughExchange and TransformDistinctInnerJoinToRightEarlyOutJoin
These optimizations are local only to identified plan sub-trees and rewrite these plan sub-trees to a minimum cost plan that they find via local exploration. The rest of the optimizer is rule based, and the unsaid assumption made in them is that all of these rules are cost reducing.

While there was an attempt [5] to reimplement the optimizer as a Cascades style [6] optimizer, there is no record of it being PRed or PoCed. Additionally, Presto includes a Cascades-style 'Memo' in the IterativeOptimizer, its implementation is inconsistent with the Cascades memo - it does not track equivalence classes (Groups) or multi-expressions which are essential to memoize previously explored sub-plans

Implementing a new cost-based optimizer

For performing a dynamic programming based Cascades style[6] plan space exploration and memoization of previously explored plans, we need a Memo with an interface like:

```
Map <KEY: Multi-Expression, VALUE: {[Logical Plans], [Physical Plans], Least_Cost_Physical_Plan>
```

Each entry in such as Map is the 'Group' to explore in the Memo.

The `KEY` in this map is a 'Multi-Expression' which is canonicalized version of the plan/sub-plan tree which uniquely identifies logically (or physically) equivalent plans.

The `VALUE` for this key is a struct with the below fields -

Field	Description	Notes
Logical Plans : List	A list Logically equivalent plans explored for this Group.	Child nodes in these plans are themselves multi-expressions, aka they are pointers to other Groups
Physical Plans : List	A list of physical implementations of all the logical plans. These have stats & costs associated with them	Interesting properties of each physical plan are also stored
Least_Cost_Physical_Plan	The cheapest physical plan identified for this Group	If a search is made into a Group with a specific property (like SortOrder), we will store more than one least cost plan with a

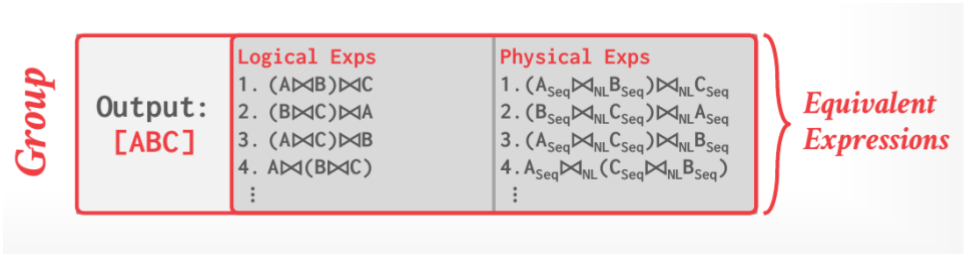
		{Property Set} that satisfies requirements

We start by seeding the Memo with the initial plan tree obtained after AST analysis.
We start at the root of the plan tree and apply a CANONICALIZE step to each of its children. This entails recursively reducing the children to a string-like representation that uniquely identifies this sub-tree. Each such unique representation identified in the recursive walk is a ‘Multi-Expression’ that we store in the Memo as an entry: `<MExpression, Ref-to-current-sub-plan>.` The parent multi-expression is a combination of the current query operator, plus the ordered list of children multi-expressions.

Once the Memo is seeded, we start from the root Group and using the `Set<LogicalCostBasedTransformationRules>` that match against this tree shape, compute and store logically equivalent plans for this Group. Note that this Logical plan refers to its children using ‘multi-expressions’, so that we can recursively explore the node’s children in the same way.

After all logical transformations for a node are complete, we then apply `Set<PhysicalTransformationRules>` to compute the possible physical plans for this Group, along with its stats & costs

An example of a Group for (`A IJ B IJ C`), would be -


Group
Output: [ABC]
Logical Exps
1. (A⋈B)⋈C
2. (B⋈C)⋈A
3. (A⋈C)⋈B
4. A⋈(B⋈C)
⋮
Physical Exps
1. $(A_{Seq} \bowtie_{NL} B_{Seq}) \bowtie_{NL} C_{Seq}$
2. $(B_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} A_{Seq}$
3. $(A_{Seq} \bowtie_{NL} C_{Seq}) \bowtie_{NL} B_{Seq}$
4. $A_{Seq} \bowtie_{NL} (C_{Seq} \bowtie_{NL} B_{Seq})$
⋮
Equivalent Expressions

Once we have explored a Node fully, we save the Least Cost physical plan identified for this ‘Group’. In this way we memoize the optimal plan for this Group; **future lookups for this ‘Group’ will reuse the already identified least cost physical plan**

Optimization based on interesting properties is also possible with this setup. For this, we would pass down a set of plan constraints (such as sort order or result distribution) that the explored children must satisfy. We would memoize the least cost plan for each set constraint-set.

The search for the least cost plan can end once we've fully enumerated all physical plans for the root Group. Since this can be inordinately long (e.g for more than 10 joins), we can instead end the search after a fixed amount of time, or if we've explored the tree to a specific depth (detailed requirements on exit conditions are out of scope of this document)

More details and walk through example of how rules would apply can be found in [7] and [8]. The paper mentioned in [9] has details on how interesting properties could be used to identify least cost plans for a group

Open Questions & Key Design considerations

The new cost-based optimizer could implement the existing PlanOptimizer interface. We will start with a small set of rules (plus add new rules) that need to be cost based. Rules can continue to use the same Pattern & Capture classes to match against multi-expressions. The pertinent questions (in order of highest to lowest priority) are –

1. Enforcing interesting orders and physical properties: See appendix
2. Performance: Storing and exploring the Memo with plan alternatives, interesting orders & properties will be very CPU and memory intensive
3. Exit rules for ending cost-based search: For complex queries, we can end up generating a large set of Groups that need exploring. There are search strategies mentioned in [1], [8] and [9] that can be used to prune the search space that we need to decide on
4. Ensuring we don't get stuck in a loop while exploring the Memo – Cycle detection while exploring the search graph is crucial. See this for examples
5. How do we generate the multi-expression for a Group so that (most) logically equivalent plans resolve to the same Group.
6. Classify existing Presto rules as logical or physical and flag those that are candidates for cost-based optimization: See appendix / this sheet
7. How should cardinality and cost-model improvements tie in: These should be computed per Memo group

Execution Plan

We can add the new CBO without requiring major changes and regressing the rest of the Presto planner. To do so we need to build a new `CascadesOptimizer : PlanOptimizer` class which implements the cascades optimizer

- The implementation (in Java, Rust, C++) that takes as input, a serialized logical Presto plan and produces an optimized physical plan
- We migrate the existing cost-based rules one by one. To start, we will migrate the existing ReOrderJoins implementation
- We will allow users to turn on this new CBO using a feature flag. Without the flag, the existing iterative-optimizer-counterpart rule will instead run (traditional heurstic rule)

End State / Exit critieria

- We have migrated/created 5 new rules for the CBO

Related projects in open source -

- See appendix

References

[1] Chaudhuri S. An overview of query optimization in relational systems. In: Proceedings of 17th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems. 1998. p. 34–43.

[2] Cost-based query transformation in Oracle, R Ahmed, A Lee, A Witkowski, D Das, H Su, M Zait, T Cruanes, VLDB, 2006

[3] <Link to Search space improvements doc>

[4] DetermineJoinDistributionType, DetermineSemiJoinDistributionType

[5] https://github.com/prestodb/presto/wiki/New-Optimizer

[6] Cascades paper, Columbia query optimizer

[7] CMU Optimizer Implementation Part2 talk

[8] CMU Optimizer Implementation Slides

[8] Prof. Andy Pavlo – Optimizer Implementation Part2 slides

[9] Orca: a modular query optimizer architecture for big data

APPENDIX

Presto existing rule classification

This sheet examines Presto iterative optimizer rules and PlanOptimizers and tags them as Logical or Physical rules, and if they are good candidates for rewriting as cost-based rules

Enforcing interesting orders and physical properties

Below is an example snapshot of how the query -

```
SELECT T1.a FROM T1, T2

WHERE T1.a = T2.b

ORDER BY T1.a;
```

Would be represented in the Memo for the requirement that T1.a is returned sorted (copied from the Orca Optimizer paper [9])

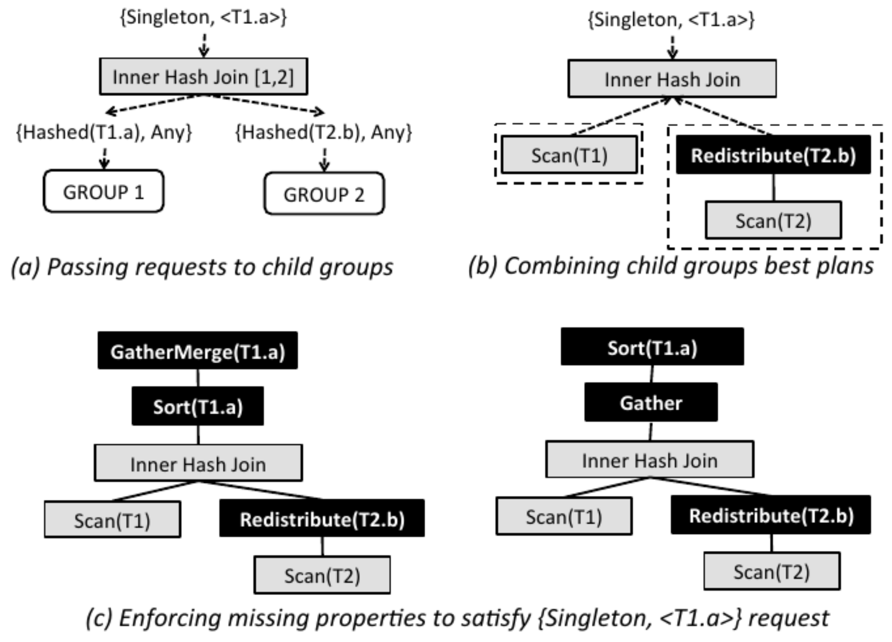
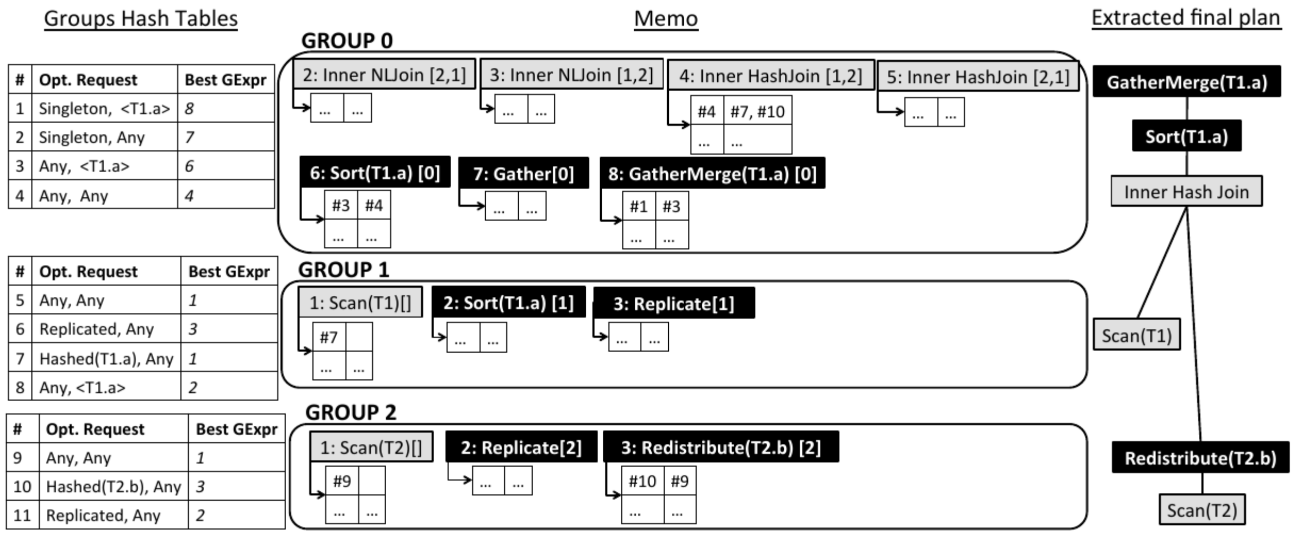


Figure 7: Generating InnerHashJoin plan alternatives



#	Opt. Request	Best GExpr
1	Singleton, <T1.a>	*8*
2	Singleton, Any	*7*
3	Any, <T1.a>	*6*
4	Any, Any	*4*

#	Opt. Request	Best GExpr
5	Any, Any	*1*
6	Replicated, Any	*3*
7	Hashed(T1.a), Any	*1*
8	Any, <T1.a>	*2*

#	Opt. Request	Best GExpr
9	Any, Any	*1*
10	Hashed(T2.b), Any	*3*
11	Replicated, Any	*2*

Figure 6: Processing optimization requests in the Memo

Example of Group By pull up correctness:

We checksum the output and prove that we are selecting the same output rows

```
presto:tpch_sf100_parquet> SELECT checksum(ROW(V.orderdate, V.receiptdate)) FROM ( select l.receiptdate, o1.orderdate, o1.totalprice,
min(o2.totalprice) OVER (PARTITION BY o2.orderkey) as avgPricePerOrderKey FROM orders o2, orders o1, lineitem
l WHERE o1.orderkey = l.orderkey and l.extendedprice > 104000 and o2.orderkey = o1.orderkey ) V WHERE V.totalprice =
V.avgPricePerOrderKey;
        _col0
-------------------------
e6 d4 37 ac 14 87 57 1e
(1 row)

Query 20231102_050240_00018_s5i8q, FINISHED, 4 nodes
Splits: 991 total, 991 done (100.00%)
[Latency: client-side: 0:14, server-side: 0:14] [900M rows, 5.25GB] [66.4M rows/s, 396MB/s]

presto:tpch_sf100_parquet> SELECT checksum(ROW(o1.orderdate, l.receiptdate)) FROM orders o1, lineitem l, ( SELECT orderkey, MIN
(o2.totalprice) as avgTotalPrice FROM orders o2 GROUP BY orderkey ) o2 WHERE o1.orderkey = l.orderkey and o1.orderke
y = o2.orderkey and l.extendedprice > 104000 and o1.totalprice = avgTotalPrice;
        _col0
-------------------------
e6 d4 37 ac 14 87 57 1e
(1 row)

Query 20231102_050403_00021_s5i8q, FINISHED, 4 nodes
Splits: 991 total, 991 done (100.00%)
[Latency: client-side: 0:19, server-side: 0:19] [900M rows, 5.25GB] [48.1M rows/s, 287MB/s]
```

Current state of art frameworks for Cost Based Transformations

We examine existing cost-based frameworks for plan transformations, state space search and plan tree memoization

Oracle

As described in the paper [2]

DuckDb

DuckDb uses a rule based optimizer. Only cost based transformations are for Join reordering

Postgres

Postgres doesn't have a cost-based optimizer either

Cascades Optimizer Implementations

CockroachDb

Memo implementation storing logically equivalent plans in a group : Memo is keyed by the 'interned' (hashed) value of the plan node Interning can be done on relational or scalar expressions The Group in a Memo is a RelExpr which is a linked-list of different equivalent multi-expressions. Each RelExpr has a cost

The optimizer operates by

1. Starting at the root of the Memo, call optimizeGroup
2. This iterates on on all the group members by calling optimizeGroupMember, which then recursively traverses this group if it can satisfy the logical/physical properties needed for this group
3. After the call to all children of optimizerGroupMembers, we do an exploreGroup/exploreGroupMember call. Its in this call that we generate alternate plans using rules for that match for this group member
4. Since CockroachDb uses codegen to generate (Go) rule code, any rules tagged with Explore are the ones that pattern matched against the member

optd - Rust

https://cmu-db.github.io/optd/datafusion.html : A ~3 month old research cascades impl in Rust

optd is designed as a flexible optimizer framework that can be used in any database systems. The meat of optd is in optd-core, which contains the Cascades optimizer implementation and the definition of key data structures in the optimization process. Users can implement the interfaces and use optd in their own database systems by using the optd-core crate.

- Tested with 3 table joins: The JoinCommuteRule and JoinAssocRule are capable of building the join tree pretty quickly
- Failing for a large 8 table join: I suspect this was due to search space explosion not being tackled correctly by the unbounded cost optimizer
- **Memory use was very low**: One of the salient features of using Rust is tight control on Memory use

databend – Rust

- Databend built their CBO from closely following the CMU optimizer course
- Built ~2021
- https://docs.databend.com/guides/overview/community/rfcs/new-sql-planner-framework
- Not extensible enough, will have to fork

https://github.com/datafuselabs/databend/blob/main/src/query/sql/src/planner/optimizer/cascades/cascade.rs
Rules : https://github.com/datafuselabs/databend/blob/main/src/query/sql/src/planner/optimizer/cascades/cascade.rs

Calcite - Java

Has a VolcanoPlanner class which holds:

- IdentityHashMap<RelNode, RelSubset> mapRel2Subset : A map of RelNode (the plan node) and what equivalence class (RelSubset) it belongs to
- Can apply rules top down like a cascades optimizer using TopDownRuleDriver - This is a new feature, added in 2020. Entry point is : #drive - GH commit : https://github.com/apache/calcite/commit/33aa61ca404018cc9fe8ad2ec2c02ba269c67ebe
- Email disussion on adding a cascades style optimizer : https://lists.apache.org/thread/fn1wwkb62byk2vlpqqsgmsllj6xjgprq

5. Branch and Bound Space Pruning

> *After we implement on-demand, top-down trait enforcement and rule-apply, we can pass the cost limit at the time of passing down required traits, as described in the classical Cascades paper. Right now, Calcite doesn't provide group level logical properties, including stats info, each operator in the same group has its own logical property and the stats may vary, so we can only do limited space pruning for trait enforcement, still good. But if we agree to add option to share group level stats between relnodes in a RelSet, we will be able to do more aggresive space pruning, which will help boost the performance of join reorder planning.*
>
> *- https://www.mail-archive.com/dev@calcite.apache.org/msg13991.html*

The main entry point into this class which kick starts the optimization is findBestExp
Full working example : https://www.querifylabs.com/blog/assembling-a-query-optimizer-with-apache-calcite, https://www.phind.com/search?cache=nog59ygh9yzm9h5lsfpkyf1g

Extension of Calcite with custom operators, rules and cost functions is doable: Calcite was meant to be used as a library, so its is very extensible

Dremio : Uses the Calcite planner

GPORCA - C++

https://github.com/greenplum-db/gporca

- DXL plan transformed to logical, then physical plan with interesting properties
- Metadata is plumbed using an interface - this provides a mechanism to get info on table, stats etc
- **Extension points are missing** - no simple way to add your own rules, plan nodes (logical or physical)
- Not a lot of commits in the last two years, not a log of engagement by VMWare