Opportunistic Physical Design for Big Data Analytics

Jeff LeFevre, Jagan Sankaranarayanan,
Hakan Hacıgümüs, Junichi Tatemura,
Neoklis Polyzotis, Michael J. Carey

SIGMOD'14

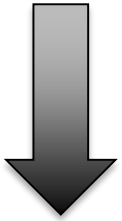
曾丹 2015-04-15

Opportunistic Physical Design?

Opportunistic Materialized Views

- In MapReduce, queries for big data analytics are often translated to several MR jobs
  - Each job outputs results to disk
  - The intermediate results are called opportunistic materialized views
- Can be reused to speed up queries
  - Exploratory queries expose reuse opportunity

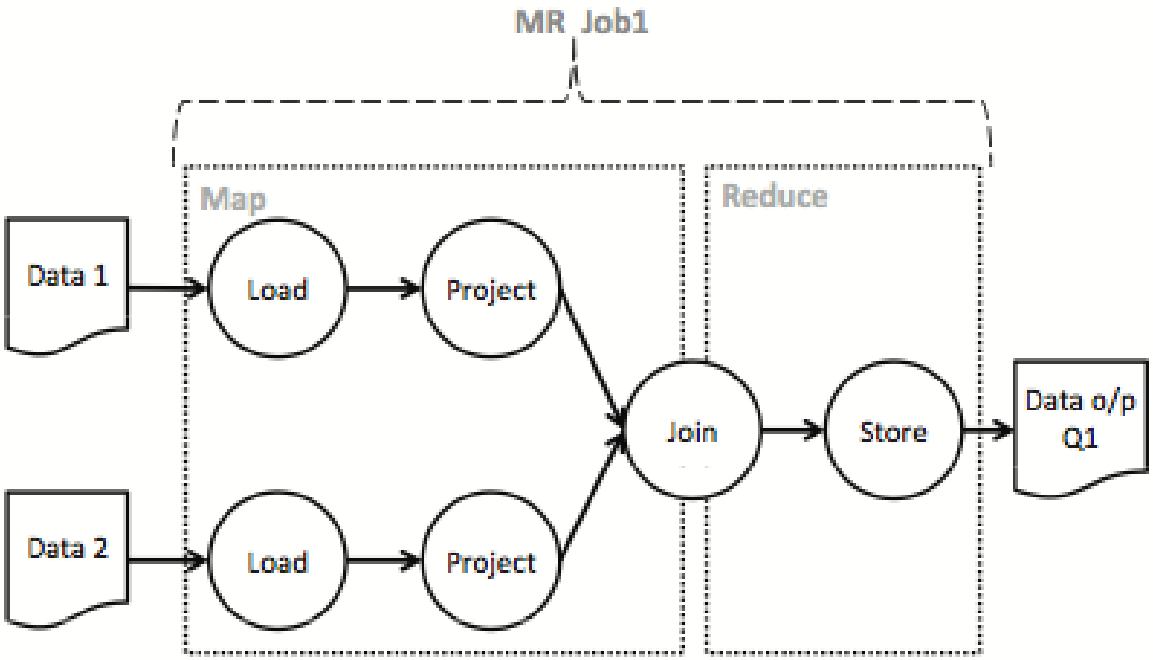
Use Opportunistic Materialized Views to Rewrite Queries



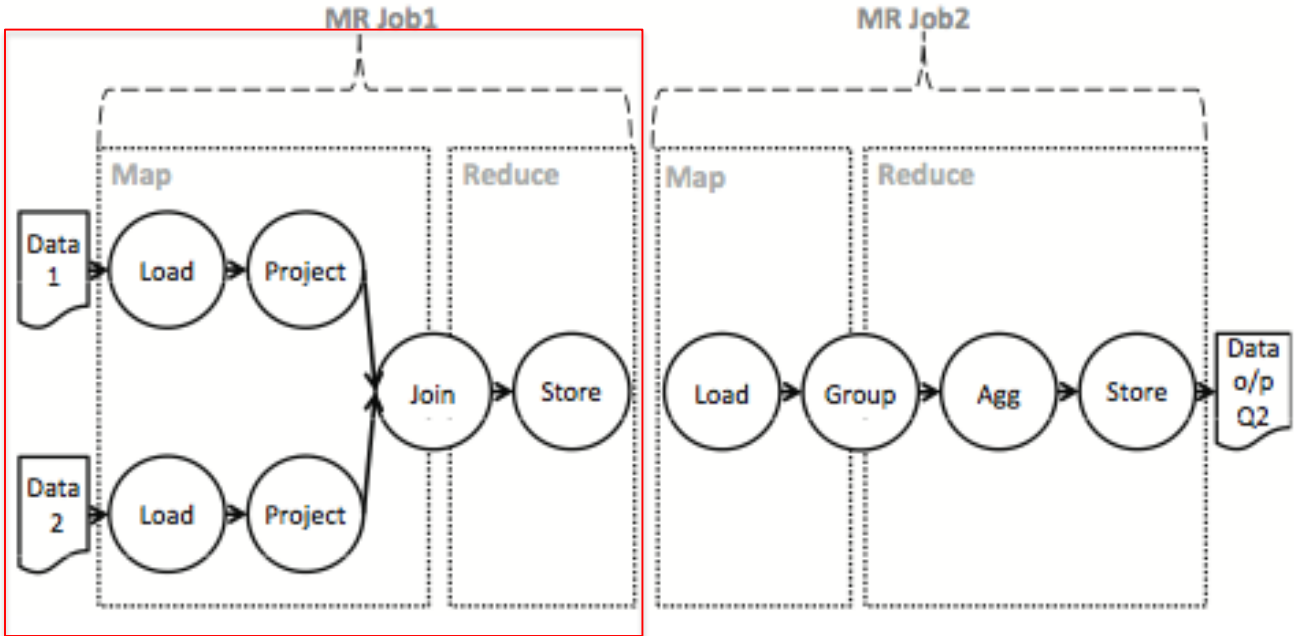
Opportunistic Physical Design

Traditional Solution

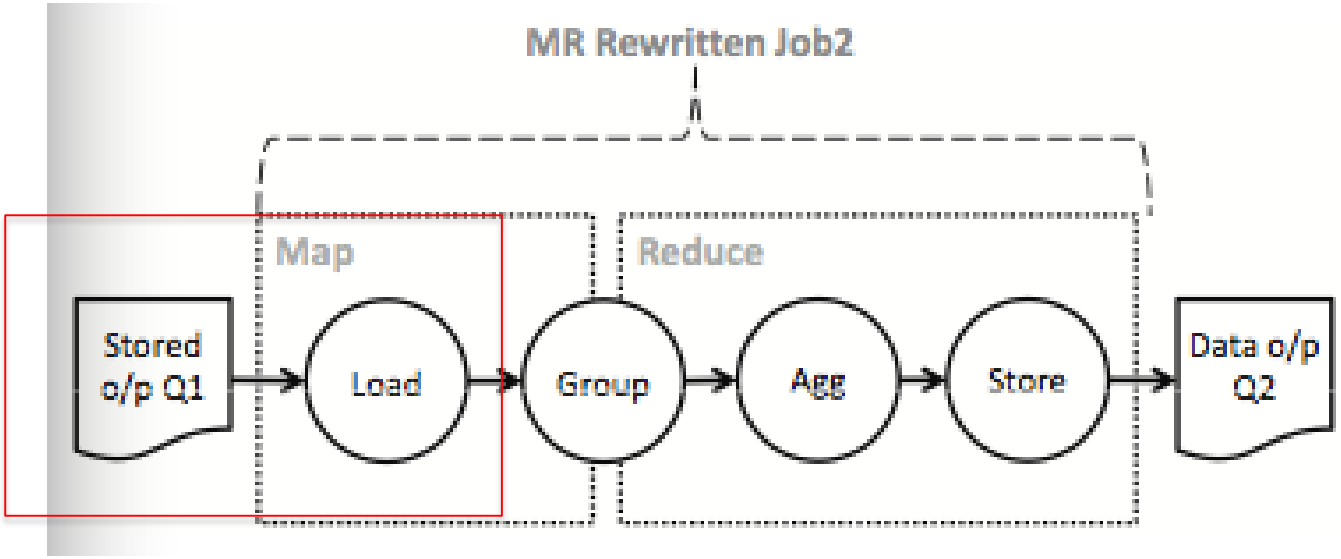
- Match query plan with the plan of view
- Replace the matched part with a load operator which loads data from view


MR Job1
Map
Reduce
Data 1
Load
Project
Data 2
Load
Project
Join
Store
Data o/p
Q1

Q1


MR Job1
MR Job2
Map
Reduce
Map
Reduce
Data
1
Load
Project
Data
2
Load
Project
Join
Store
Load
Group
Agg
Store
Data
o/p
Q2

Q2



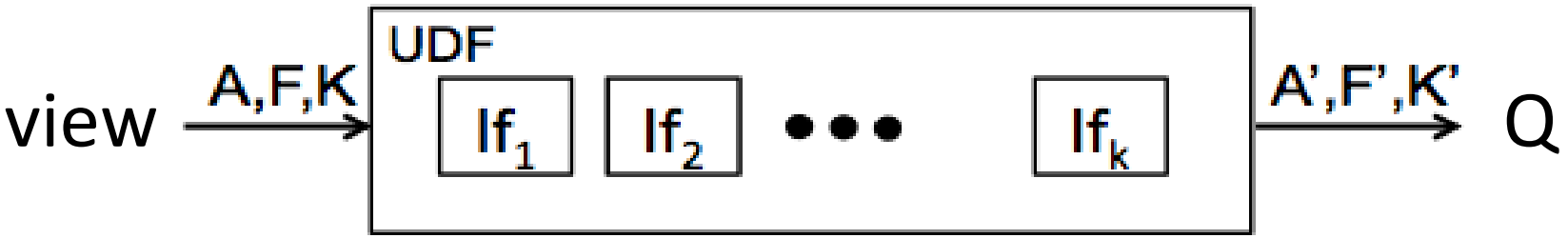
Q2 rewritten using Q1

Problems

- Can only reuse results when execution plans are identical
- In the context of MR, queries always contain UDFs
  - Hard to match udf
  - Need to understand UDF semantic => UDF Model

Rewrite Overview

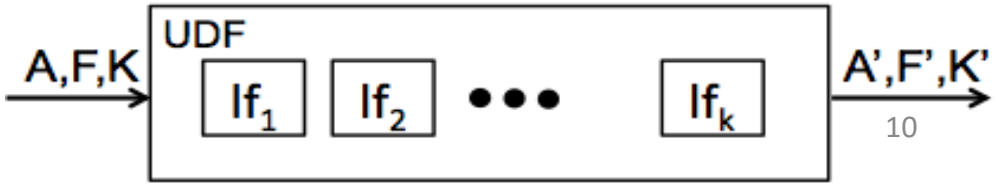
- Find candidate views
  - Match metric: UDF Model
- Use operators to define a UDF



Many solutions —Cost model→ Shrink the search space

UDF Model

- Input(A, F, K)
  - A(Attributes), F(Filters previously applied to the input), K(current grouping keys of the input)
- Output(A', F', K')
- Signature
- A composition of local functions
  - Local function represent map or reduce task
    - Discard or add attributes
    - Discard tuples by filters
    - Grouping tuples on a common key



Example

```
// T1 is input table, T2 is output table
// user_id, tweet_text are input attributes, threshold is a UDF parameter
// sent_sum is an output attribute whose dependencies are recorded
UDF_FOODIES (T1, T2, user_id, tweet_text, threshold) {
    CREATE TABLE T2 (user_id, sent_sum) FROM T1                    (a)
       MAP user_id, text USING "hdfs://udf-foodies-lf1.pl"
                        AS user_id, sent_score  CLUSTER BY user_id
       REDUCE user_id, sent_score, threshold USING "hdfs://udf-foodies-lf2.pl"
                        AS (user_id, sent_sum)
}
```

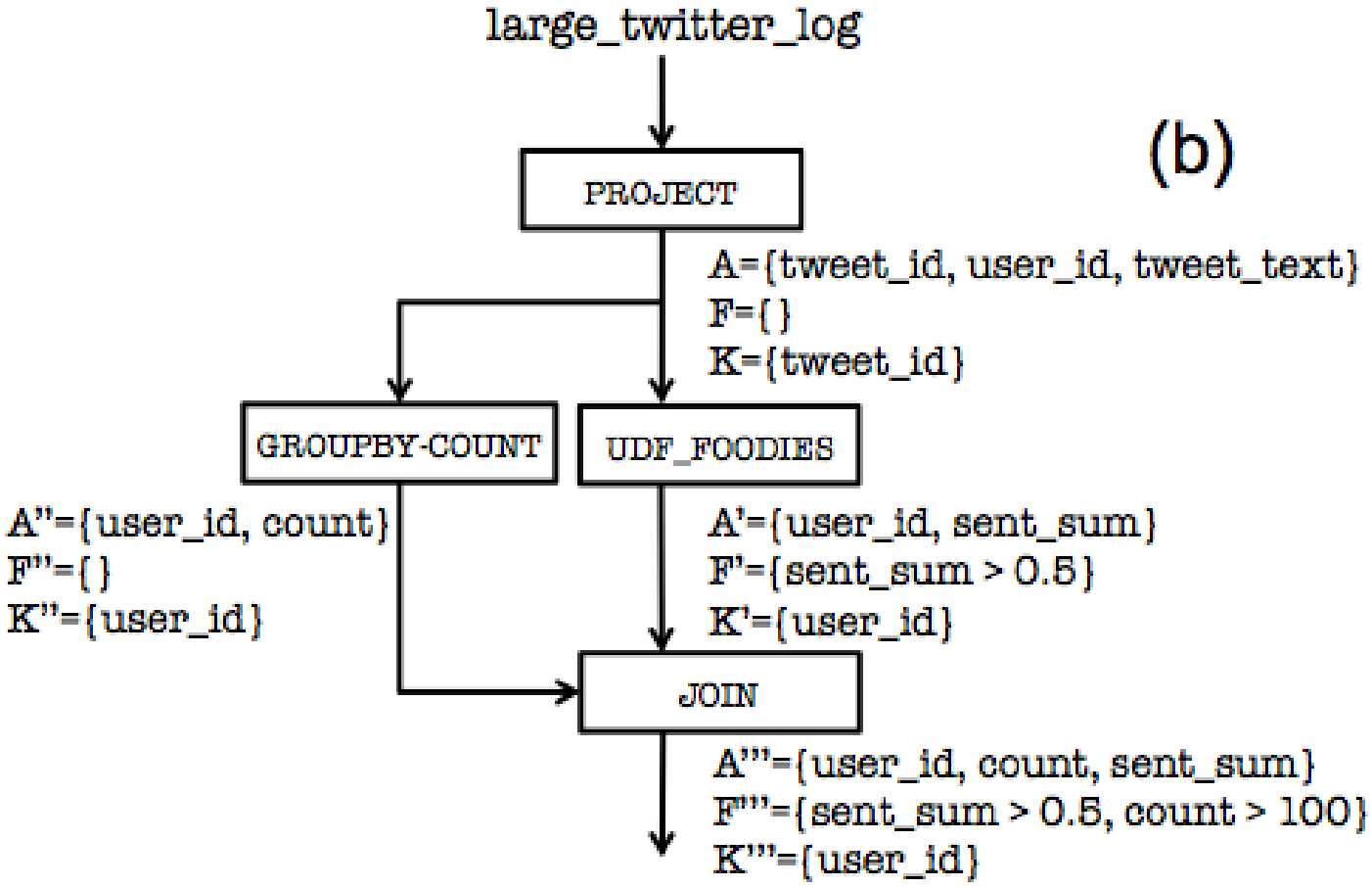
UDF model for UDF_FOODIES: (b)

A={user_id, tweet_text, ...}, F= {f}, K = {k}

A'={user_id, sent_sum}, F' = {f} ∪{sent_sum > threshold}, K' = {user_id}

Sig. of new attribute sent_sum = {UDF_FOODIES, user_id, tweet_text,{f},{k}}

```
CREATE TABLE Result
    SELECT T2.user_id, Foo.count, T2.sent_sum FROM T2,
        (SELECT user_id, COUNT(*) AS count FROM T1
            GROUP BY user_id) AS Foo
    WHERE Foo.user_id = T2.user_id AND Foo.count > 100;
```



Candidate View

- $V(A_v, F_v, K_v)$ is a candidate view of $Q(A_q, F_q, K_q)$
  - A_q is subset of A_v
  - F_v is weaker than F_q
  - V is less aggregated than Q
- Evaluate candidate views in udf cost increasing order

UDF Cost Model

- Sum of local functions cost
  - Local function with one operation
    - Cm + Cs + Ct + Cr + Cw
    - Model the baseline cost(BCm,BCr) of three operation types, Cm = x*BCm, Cr = y*BCr
    - The first time the udf is added to the system, execute the udf on a 1% uniform random sample of the input data
      - recalibrating Cm, Cr when udf is applied to new data
      - A better sampling method if more is known about data
      - Periodically updating Cm, Cr after executing the udf on the full dataset

UDF Cost Model

- Sum of local functions cost
  - Local function with several operations
    - Requires knowing how the different operations actually interact with one another
    - Provide a lower-bound

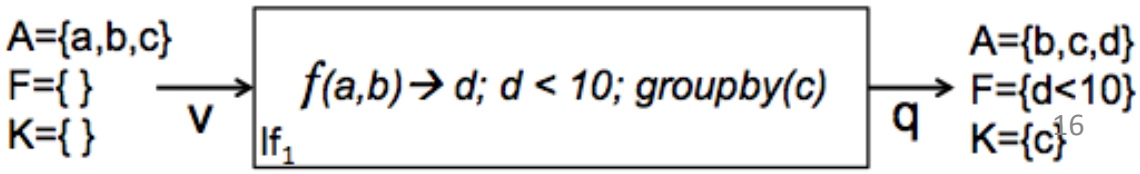
$$\textsc{Cost}(S, D) \geq \min(\textsc{Cost}(x, D), \forall x \in S)$$

Lower-bound on Cost of a Potential Rewrite

- Synthesize a hypothetical udf comprised of a single local function
  - The cost of the function is cost of its cheapest operation
- The cost of the udf represents the lower bound for any valid rewrite r

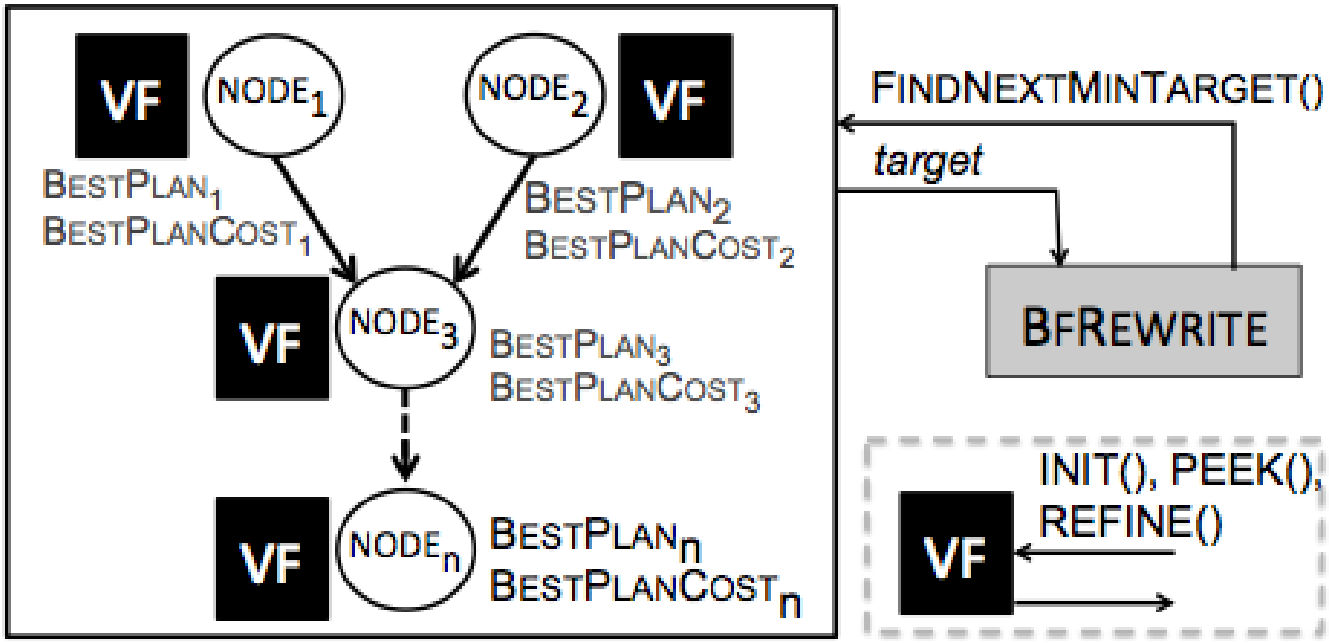
$$\textsc{OptCost}(q, v) \leq \textsc{Cost}(r)$$

- When v is not a candidate view of q, OPTCOST(q,v) = ∞



Rewrite Algorithm

- Search rewrite for each node in the query plan
  - The optimal rewrite for W_n may be worse than (optimal rewrite for $W_i + W_{i+1} \sim W_n$)



ViewFinder

- Each node has an instance VF
- A Priority Queue
  - (view, OPTCOST(Q, view))
  - Lower OPTCOST has a higher priority
- INIT
  - Initialize the queue
- PEEK
  - Get the OPTCOST of the peek element
- REFINE
  - Get rewrite r of q with the top view
  - Enumeration of operators

Rewrite Algorithm

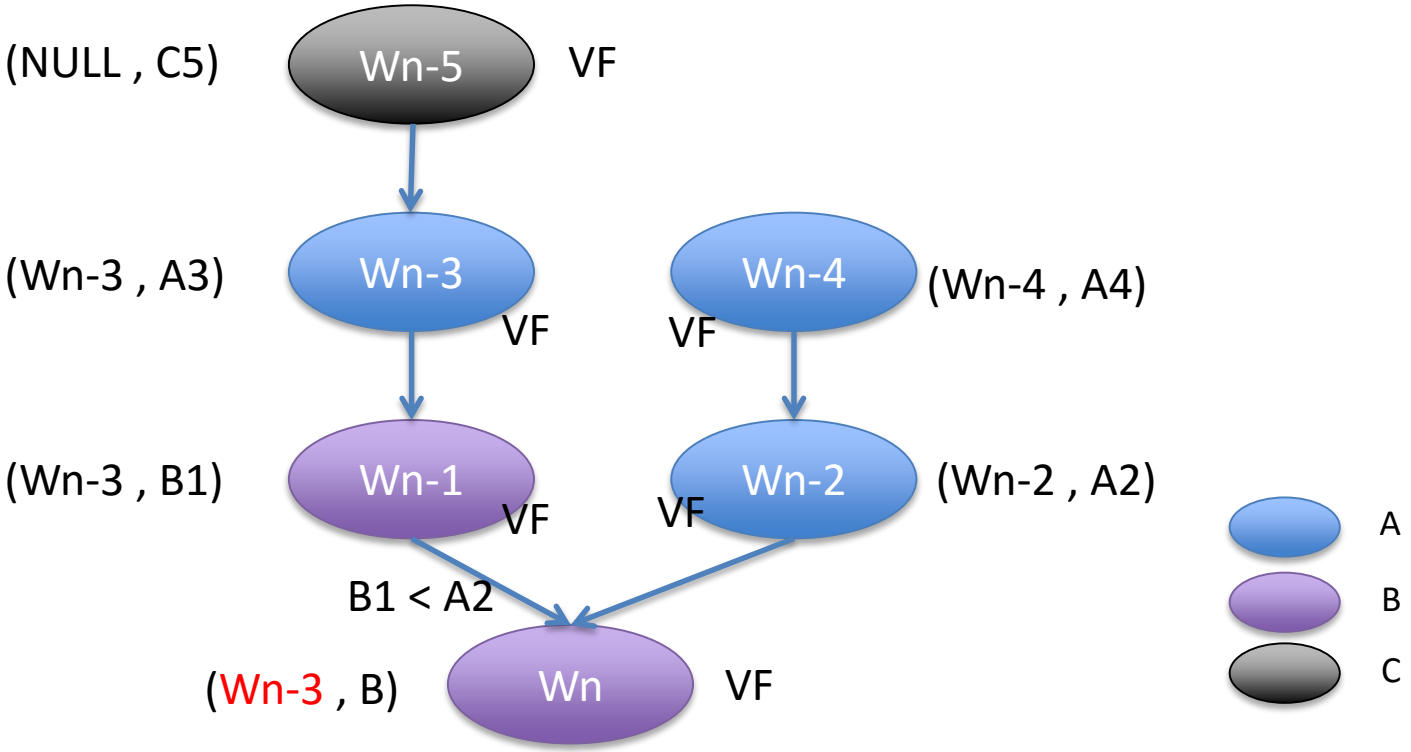
Algorithm 1 Optimal rewrite of W using VIEWFINDER

```
1:  function BFREWRITE(W, V)
2:      for each W_i ∈ W do                          ▷ Init Step per target
3:          VIEWFINDER.INIT(W_i, V)
4:          BESTPLAN_i ← W_i                         ▷ original plan to produce W_i
5:          BESTPLANCOST_i ← COST(W_i)               ▷ plan cost
6:      end for

7:      repeat
8:          (W_i, d) ← FINDNEXTMINTARGET(W_n)
9:          REFINETARGET(W_i) if W_i ≠ NULL
10:     until W_i = NULL                             ▷ i.e., d > BESTPLANCOST_n
11:     Return BESTPLAN_n as the best rewrite of W
12: end function
```

FindNextMinTarget(W_i)

- A = OPTCOST(W_i) vs B=sum($cost_{child}$) + Cost(i)
  vs C = BESTPLANCOST(i)
- Return (W_i, A) or (W_{child_min}, B) or (NULL, C)



REFINETARGET(Wn-3)

- Wn-3.ViewFinder. REFINE
  - Enumerate operators to get rewrite r
- Update the BESTPLANCOST and BESTPLAN of the upstream nodes of Wn-3

Termination Condition

- Repeat FINDNEXTMINTARGET(Wn) until it returns (NULL, cost)
- Indicate that BESTPLANCOST stored in Wn is the optimal solution

Evaluation

- Query Workload
  - From [1] contains 32 queries on three datasets that simulate 8 analysts A1-A8
    - Twritter log(TWTR), foursquare log(4SQ), landmarks log(LAND)
  - Each analyst poses 4 versions of a query
  - Executing the queries with Hive created 17 opportunistic materialized views per query on average
  - Query representation: $A_i v_j$

[1] J.LeFevre,J.Sankaranarayanan,H.Hacıgümüş,J.Tatemura,and N. Polyzotis. Towards a workload for evolutionary analytics. In *SIGMOD Workshop on Data Analytics in the Cloud (DanaC)*, 2013.

Evaluation

- Environment and DataSet
  - A cluster of 20 machines, each node has 2 Xeon 2.4GHz CPUs(8 cores), 16GB of RAM, 2TB SATA
  - Hive 0.7.1, Hadoop 0.20.2
  - 1TB of data that includes 800GB of TWTR, 250GB of 4SQ, 7GB of LAND
- Evaluation scenarios
  - Query evolution(one user)
  - User evolution(similar uses)

Evaluation

- Metric
  - Total time
    - ORIG: original execution time of the query
    - REWR: execution time of the rewritten query
  - Different algorithm of rewriting query
    - DP: searches exhaustively for rewrites at every target
    - BFR: use OPTCOST
    - Metric: time, number of candidate views examined, number of rewrites attempted
  - Comparison with caching-based methods

Query Evolution

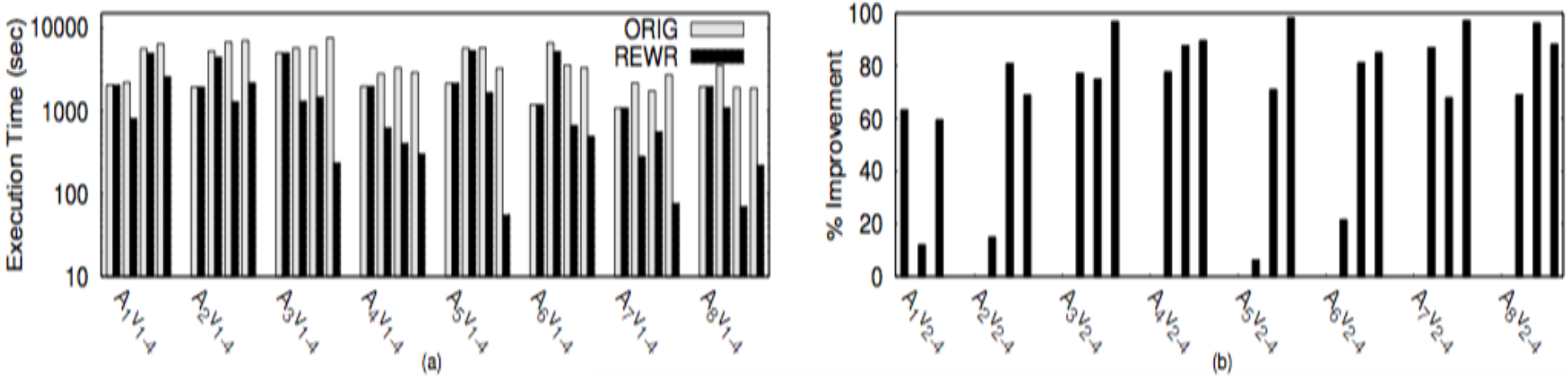


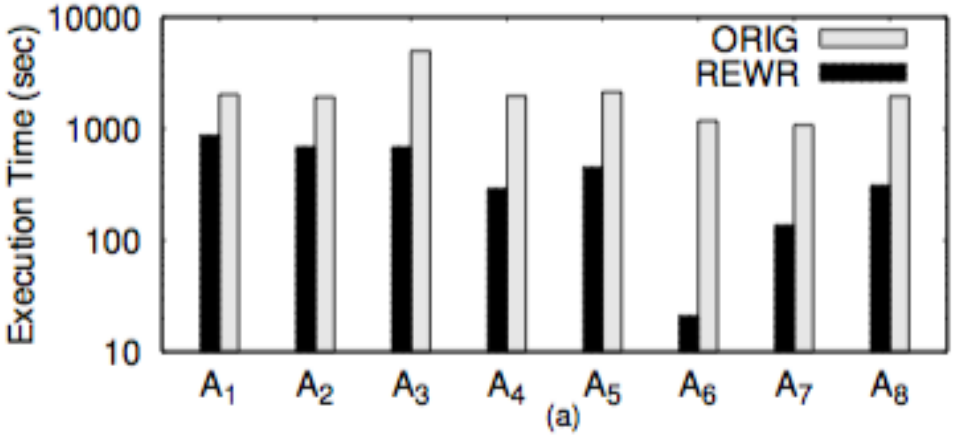
Figure 7: Query Evolution comparisons for (a) execution time (log-scale), and (b) execution time improvement.

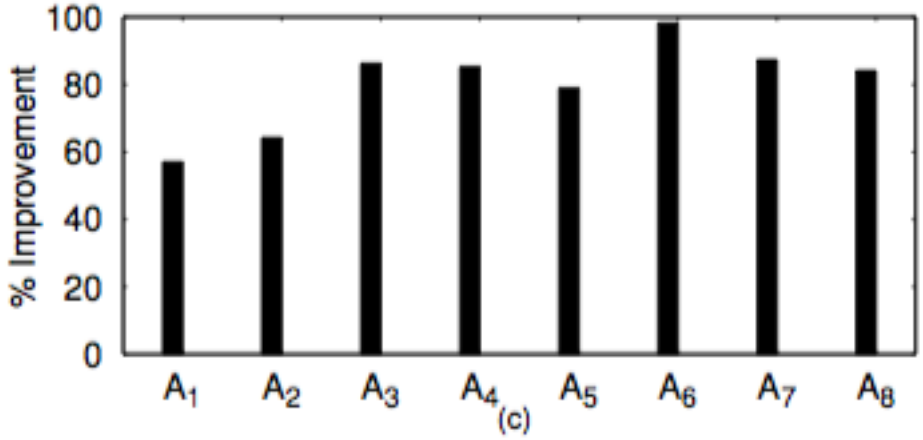
REWR provides an overall improvement of 10% to 90%, with an average improvement of 61%

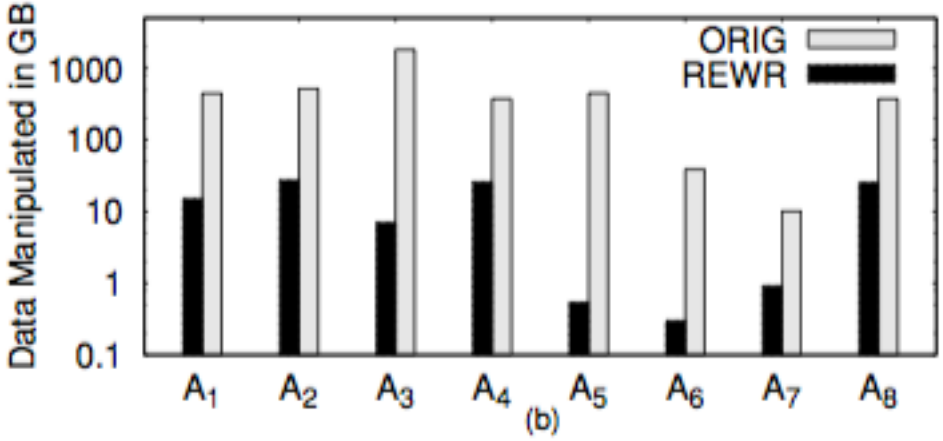
User Evolution

- A holdout analyst and 7 other analysts
- 7 other analysts execute the first version, then the holdout execute its first version, record the time
- Drop all the views and change the holdout analyst

User Evolution







REWR takes less time and manipulates less data

Overall improvement of about 50%-90%

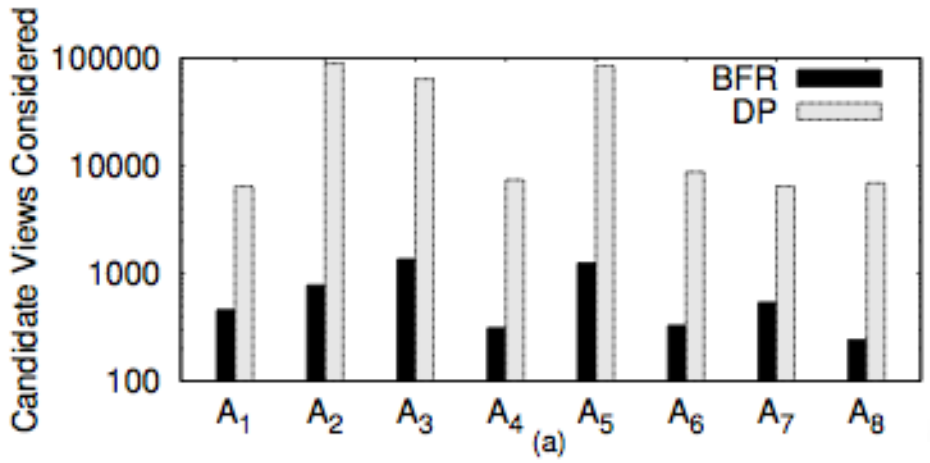
User Evolution

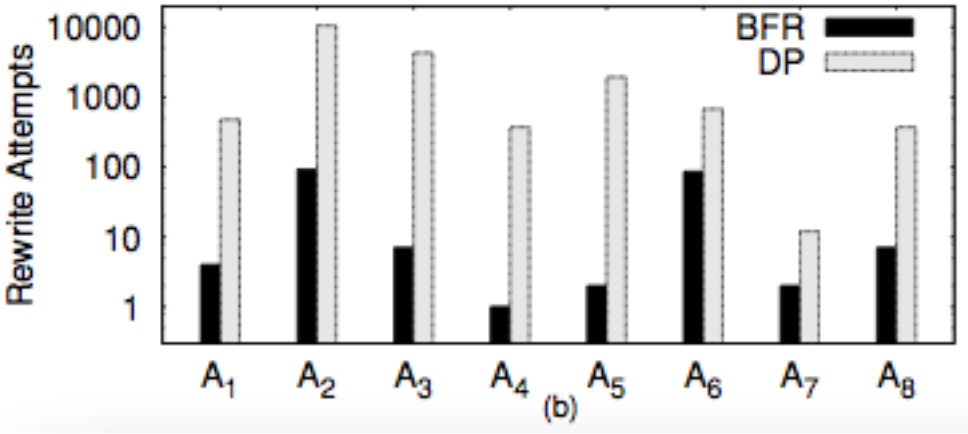
- First execute A_5v_3 as the baseline
- Gradually add analyst and execute

Table 1: Improvement in execution time of A_5v_3 as more analysts become present in the system.

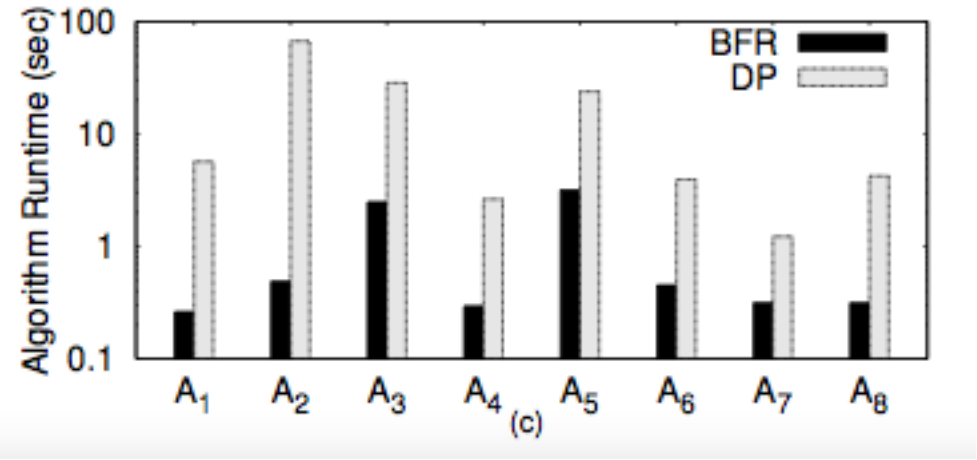
Analysts added	1	2	3	4	5	6	7
Improvement	0%	73%	73%	75%	89%	89%	89%

Algorithm Comparisons



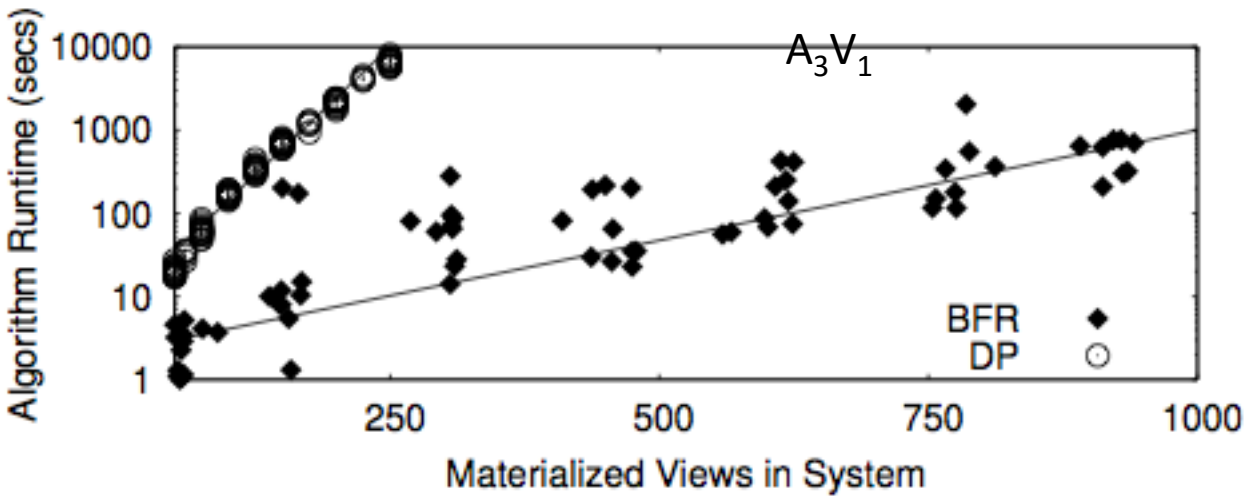


User Evolution



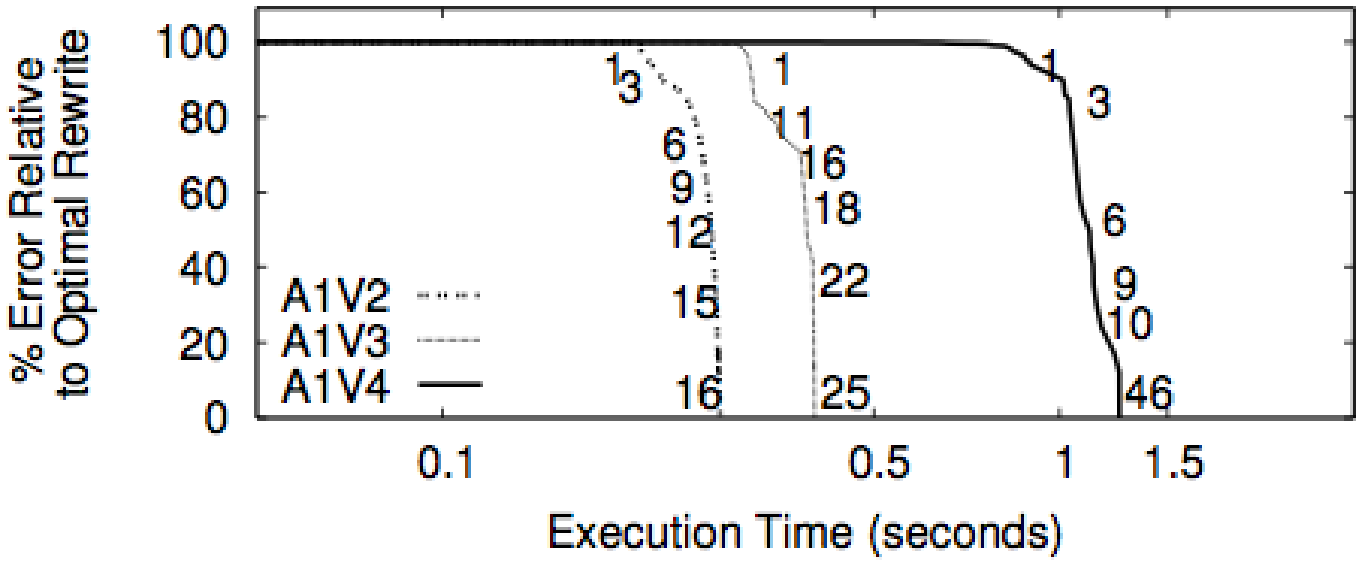
BFR narrows the search space due to GUESSCOMPLETE and OPTCOST, thus reduce the execution time

Algorithm Comparisons



BER has better scalability

Algorithm Comparisons



Once BER finds the first rewrite, it quickly converges to the optimal rewrite
The rewrite number is much smaller than DP(66, 323, 4656)

Comparison with Caching-based methods

- Identical A,F,K properties as well as identical plans

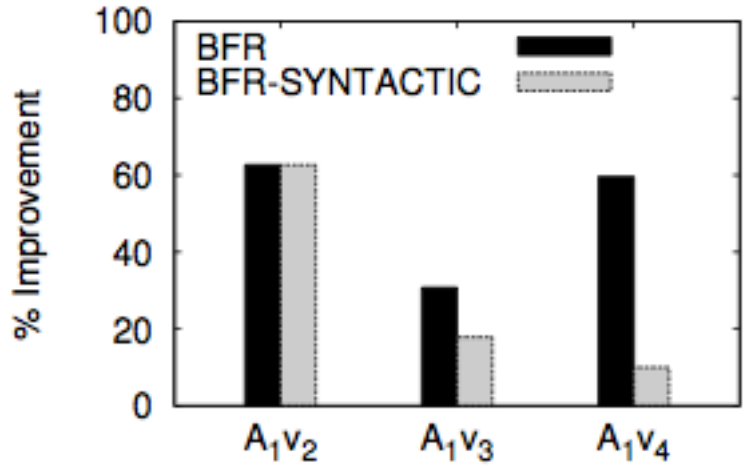


Figure 12: Execution time improvement over ORIG.

BFR has more reuse opportunity

Comparison with Caching-based methods

- Identical A,F,K properties as well as identical plans

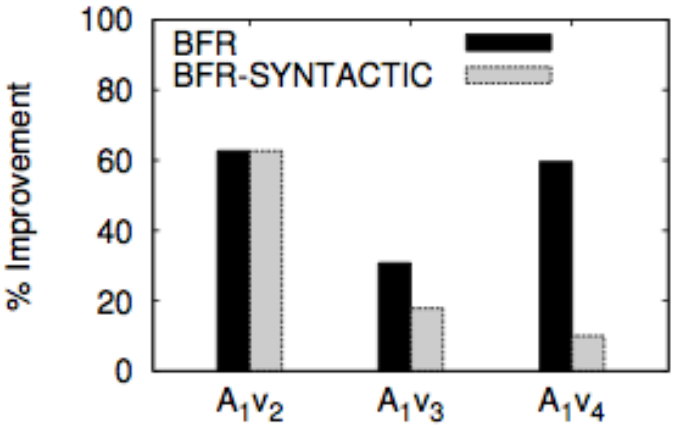


Figure 12: Execution time improvement over ORIG.

BFR has more reuse opportunity

User evolution and discard identical views

Table 2: Execution time improvement without identical views.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
BFR	57%	64%	83%	85%	51%	96%	88%	84%
BFR-SYNTACTIC	0%	0%	0%	0%	0%	0%	0%	0%

Related Work

- Traditional database area
  - Only considered restricted operator sets(SPJ/SPJGA)
  - Determine containment first and then apply cost-based pruning
- MapReduce Framework
  - Incremental computations, sharing computations or scans, re-using previous results
  - Our work subsumes these methods

Related Work

- Online physical design tuning
  - Adapt physical configuration to benefit a dynamically changing workload by actively creating or dropping indexes/views
  - Views is by-products of MR, but view selection is also needed to retain only beneficial views
- Multi-query optimization
  - Maximize resource sharing for concurrent queries

Conclusion

- A gray-box UDF model to quick find candidate view and provides a lower-bound of a rewrite
- An efficient rewriting algorithm using OPTCOST