FAST: Fragment Assisted Storage for efficient query execution in read-only databases

- **Vivek Hamirwasia** (IIIT Hyderabad), **Kamalakar Karlapalem** (IIIT Hyderabad), **Satyanarayana R Valluri** (Oracle)

Background

Row Store

Row 1	10001
	1973-09-02
	Georgi
	Facello
	M
	1996-06-26
	50000
	111111111
	9040
Row 2	10002
	1984-06-02
	Bezalel
	Simmel
	F
	1995-11-21
	60000
	222222222
	9057
...	...

High number of cache misses

Column Store

emp_no	10001
	10002
	10003
	...
birth_date	1973-09-02
	1984-06-02
	1979-12-03
	...
first_name	Georgi
	Bezalel
	Parto
	...
...	...

'row-stitching' cost is significant for high number of attributes

In-Memory Databases

- Redis
- SAP Hana
- MonetDB

Design and layout of data is the key for these systems

Contributions of our work

Provide a read-optimized hybrid storage model with algorithms to

- Co-locate attributes that are accessed together into vertical fragments
- Given a fixed amount of main memory space, allocate these fragments efficiently
- Given a query, choose those fragments that minimize the average response time for the query

Relatively easy to implement on top of existing DBMS due to low coupling.

Architecture

Relation

emp_no	birth_date	first_name	last_name	gender	hire_date	salary	SSN	manager_id
10001	1973-09-02	Georgi	Facello	M	1996-06-26	50000	111111111	9040
10002	1984-06-02	Bezalel	Simmel	F	1995-11-21	60000	222222222	9057
10003	1979-12-03	Parto	Bamford	M	1996-08-28	40000	333333333	9014
...	...	...	...	...	...	...	...	...

Secondary Storage

k	emp_no
1	10001
2	10002
...	...

(a) EMP_1

k	birth_date
1	1973-09-02
2	1984-06-02
...	...

(b) EMP_2

k	first_name
1	Georgi
2	Bezalel
...	...

(c) EMP_3

...

k	contact
1	6507968303
2	6507983837
...	...

(d) EMP_{10}

Main Memory

k	emp_no	salary
1	10001	70000
2	10002	60000
...	...	...

(a) f_1

k	gender	salary
1	F	70000
2	M	60000
...	...	...

(b) f_2

k	emp_no	last_name	hire_date
1	10001	Facello	1996-06-26
2	10002	Simmel	1995-11-21
...	...	...	...

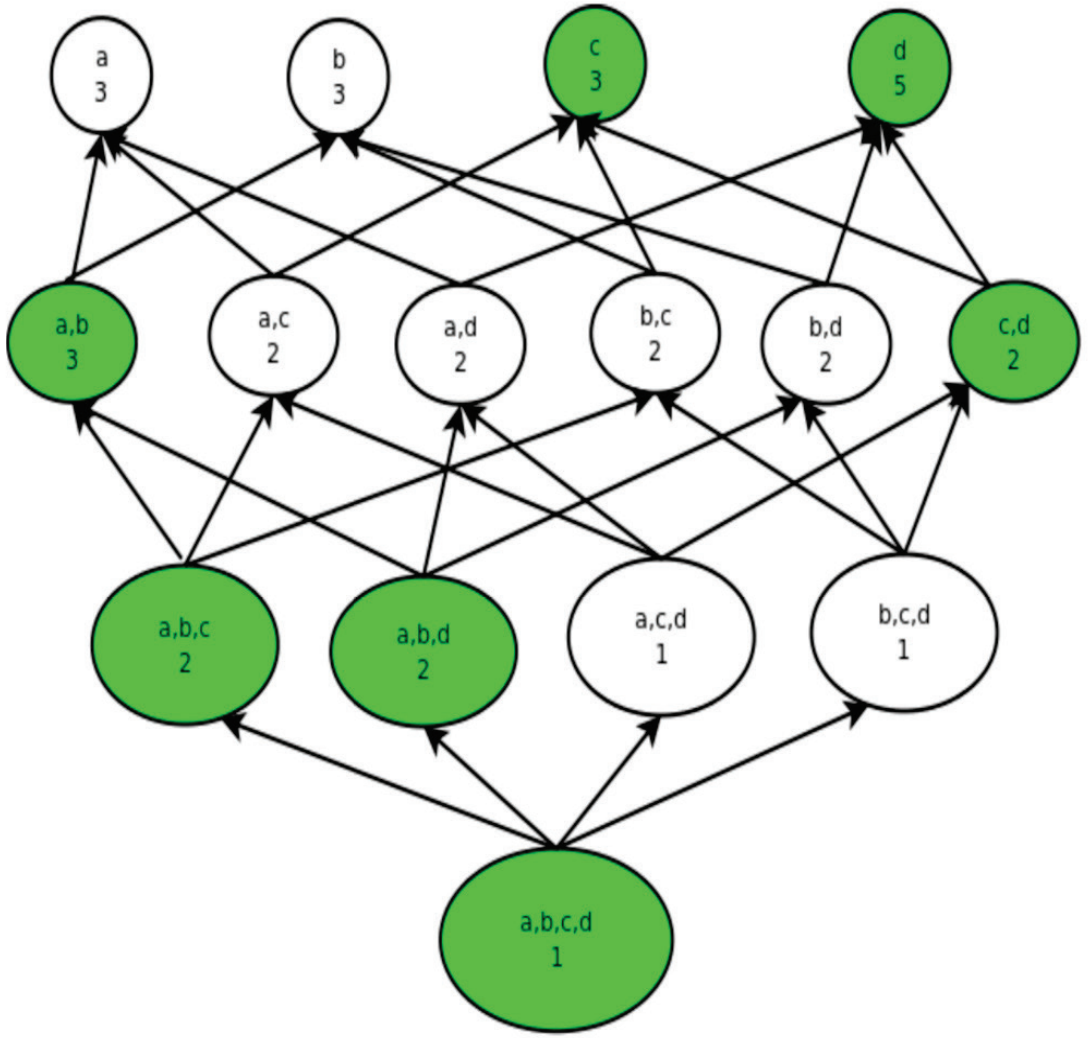
(c) f_3

Fragment generation

Possible number of vertical fragments is exponential – impractical to exhaust the search space

Frequent Closed Itemset Mining

- Itemset is frequent (i.e support count is at-least X)
- No super-set with equal or higher support count
- DBV-Miner algorithm

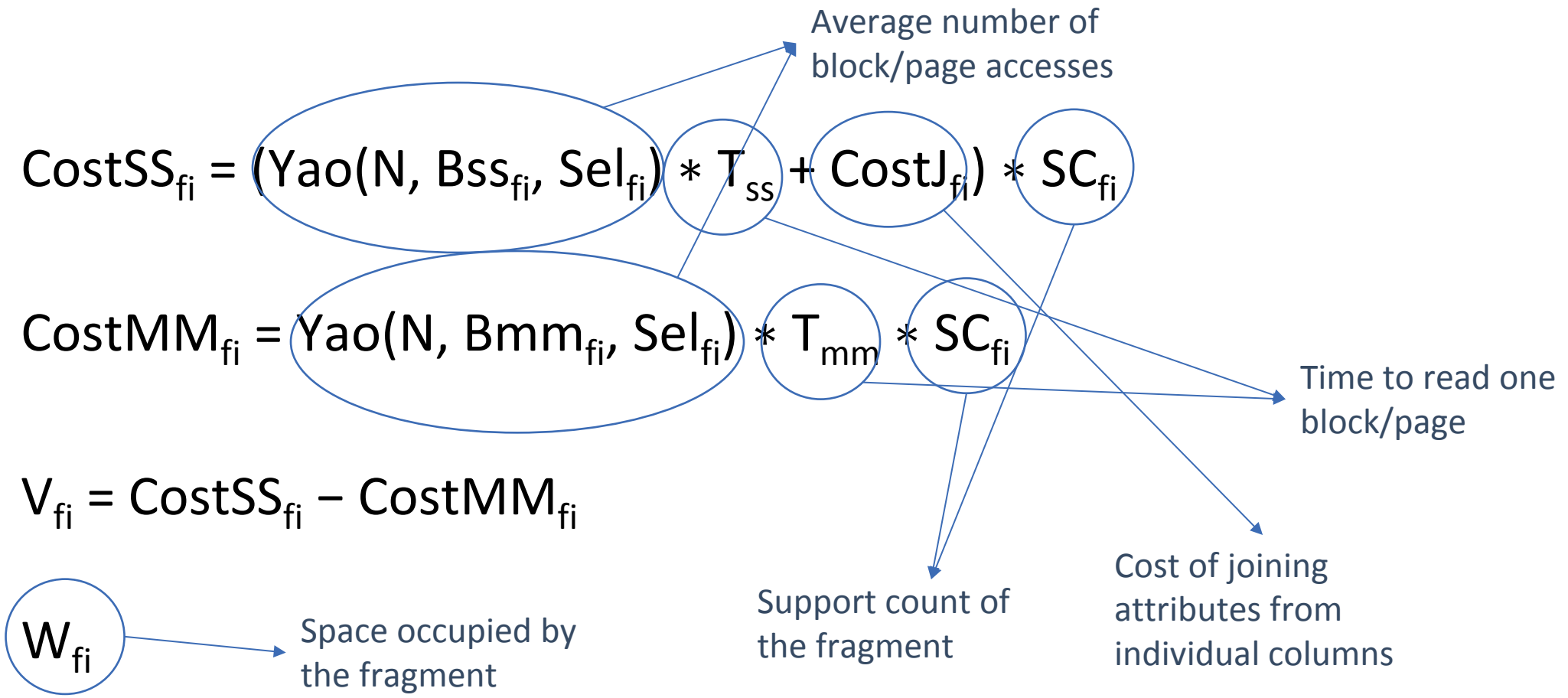


Fragment allocation

$$CostSS_{fi} = (Yao(N, Bss_{fi}, Sel_{fi}) * T_{ss} + CostJ_{fi}) * SC_{fi}$$

$$CostMM_{fi} = Yao(N, Bmm_{fi}, Sel_{fi}) * T_{mm} * SC_{fi}$$

$$V_{fi} = CostSS_{fi} - CostMM_{fi}$$



Goal: Maximize sum of values with the constraint that sum of weights is not more than M

Fragment allocation - continued

When fragments are mutually disjoint

- Similar to 0/1 knapsack which runs in pseudo-polynomial time
- To completely utilize M, we allow *fractional* fragments
- Reduces to the problem of fractional knapsack, which chooses fragments with higher value-to-weight ratio in a greedy manner

When fragments have overlap

- $SC_{f(i,j)} \geq SC_{fi} + SC_{fj}$, f(i,j) is a closed itemset!
- Use this observation to remove f(i,j) from the two fragments, resulting in three disjoint fragments
- Apply the algorithm for the mutually disjoint case

Incorporating Indexes

$$Cost(I_j^{ss}) = B(I_j^{ss}) * T_{ss} * SC(I_j)$$

$$Cost(I_j^{mm}) = B(I_j^{mm}) * T_{mm} * SC(I_j)$$

$$V(I_j^{mm}) = Cost(I_j^{ss}) - Cost(I_j^{mm}) \text{ , } W(I_j)$$

Sort fragments and indexes by their value-to-weight ratio and choose the best in each iteration until main memory buffer is full

Query reformulation & execution

Data Source Selection

- Goal: select the right set of fragments from the main memory and columns from disk
- Cost associated with each “data source” – dependent on I/O and cache processing
- If number of attributes in Q (say p) is small (i.e p < 20), do a quick exhaustive search for all combinations in 2^p
- Else
  - consider each data source to be a set of items
  - attributes in Q to be the target set
  - reduces to “weighted set cover” problem - greedy approximation in polynomial time

Query reformulation & execution

Query Reconstruction

SELECT first_name , last_name , gender

FROM EMP
WHERE salary >= 50000

SELECT EMP_3.first_name, f_3. last_name , f_2. gender

FROM EMP_3, f_3, f_2

WHERE salary >= 50000 **AND** EMP_3.k = f_3.k

Query Optimization

- Late materialization – operate on *bit-vectors* returned by indexes and materialize only when producing the result or performing a join
- Delegate the query optimization to a Selinger-style optimizer that uses dynamic programming to determine the order of joins

Workload-aware reorganization

Goal : Dynamically adapt the main memory to the changes in workload

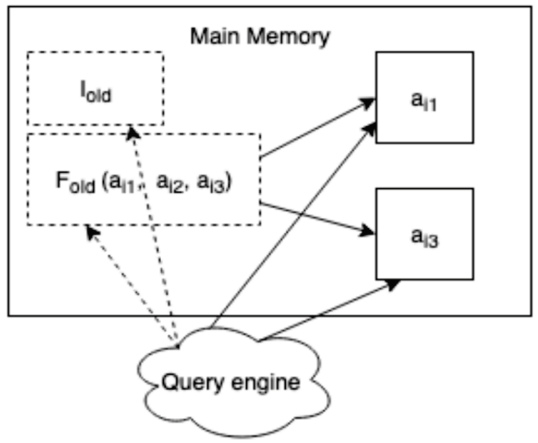
Two possible strategies:

1. Periodically generate and allocate fragments after some time
    - Expensive
    - Blocks query execution

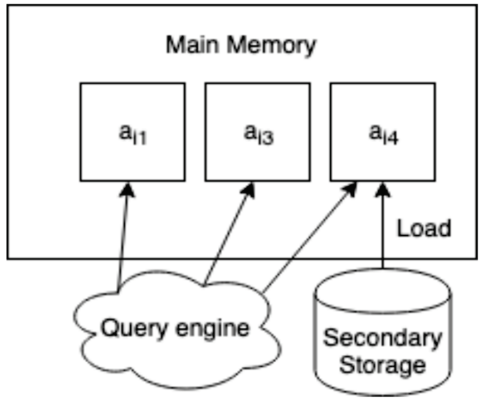
2. Detect changes in workload and incrementally reorganize main memory
    - Amortize cost over several queries
    - Can be done in a background process -> does not block query execution

Workload-aware reorganization - continued

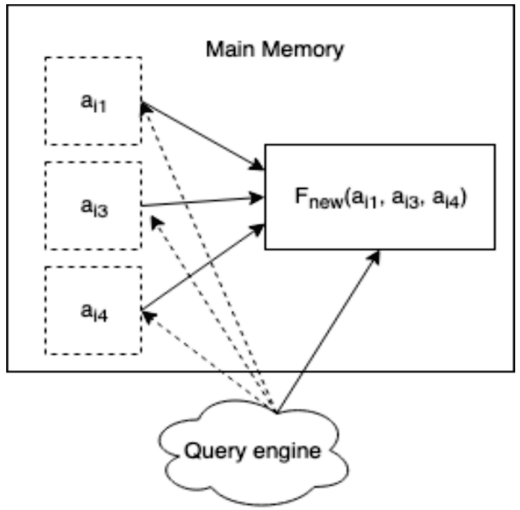
- Use a lightweight logging framework to measure main memory & disk usage to detect change in workload
- Actual reorganization happens incrementally in four independent stages



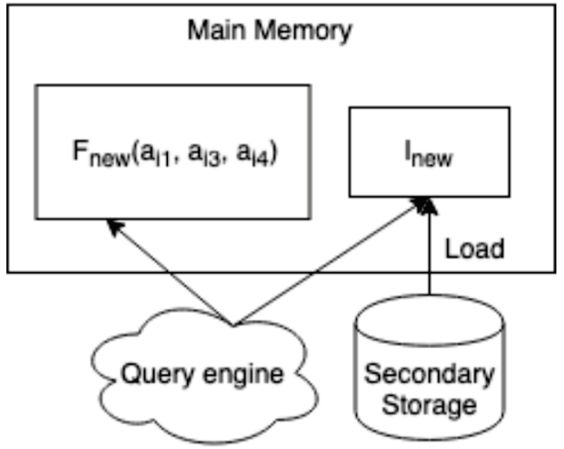
Split



Load



Merge

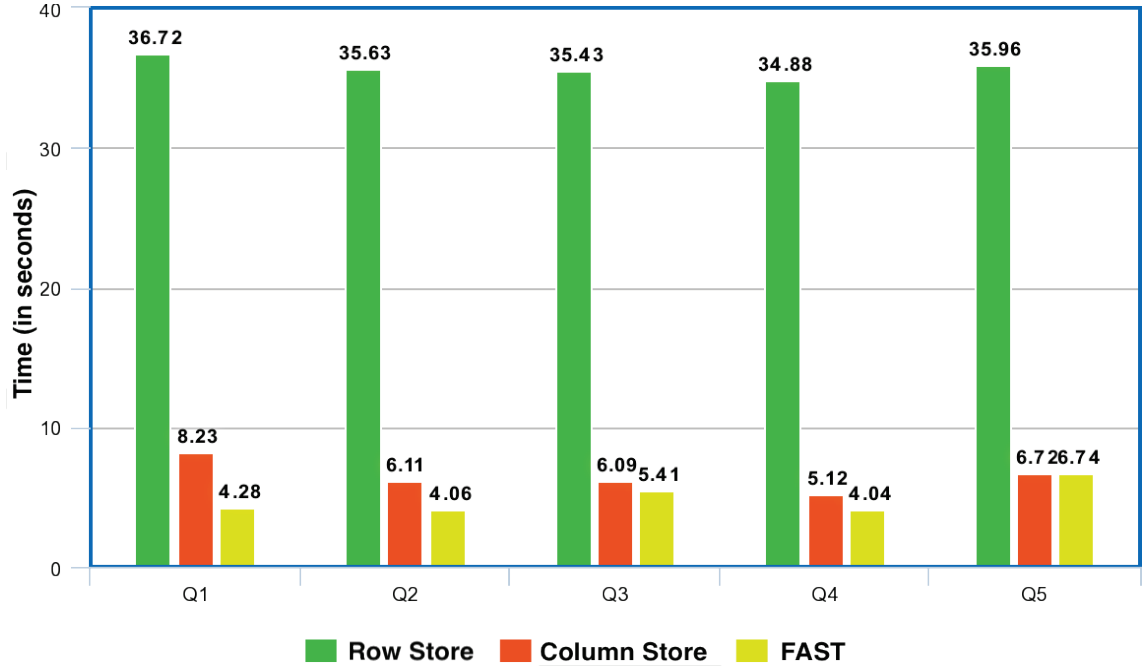


Load/Build index

Experiment: Custom benchmark

Single relation A with 50 integer attributes, selectivity of queries is 1%.

- 30% Q1: **SELECT** AVG(a0 +a1 +a2 +a3 +a4 +a5 +a6) **FROM** A **WHERE** a7 < X
- 20% Q2: **SELECT** AVG(a5 +a6 +a8 +a9 +a10) **FROM** A **WHERE** a7 < X
- 20% Q3: **SELECT** AVG(a11 +a12 +a13 +a14 +a15) **FROM** A **WHERE** a16 < Y
- 20% Q4: **SELECT** AVG(a15 +a17 +a18 +a19) **FROM** A **WHERE** a16 < Y
- 10% Q5: Remaining queries operating on ≤ 6 random attributes between a20 and a50

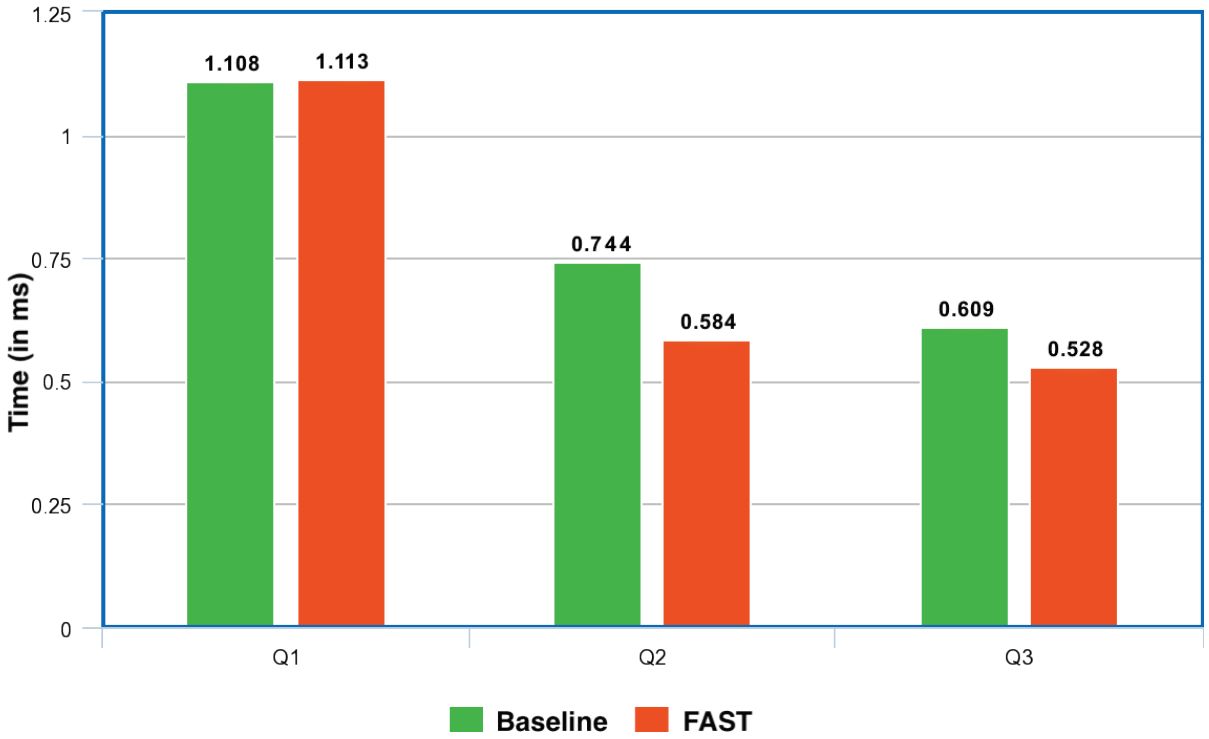


F_1(a5, a6), F_2(a15) and F_3(a0, a1) allocated by FAST

More than 7 times faster than row store and 1.3 times faster than column store

Experiment: Analyzing cache efficiency

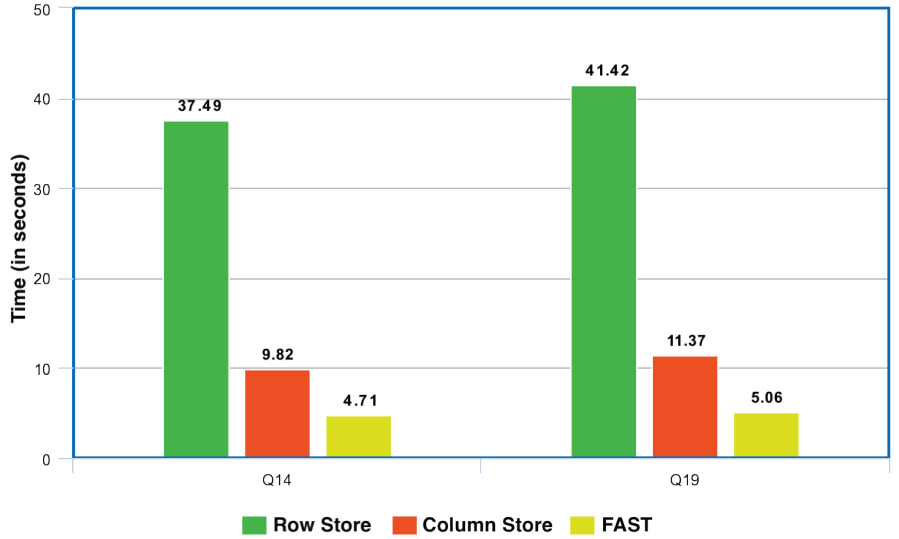
Compare fragments allocated by FAST to a baseline fragment: $F_{baseline}$(a0, a1, a5, a6, a15)

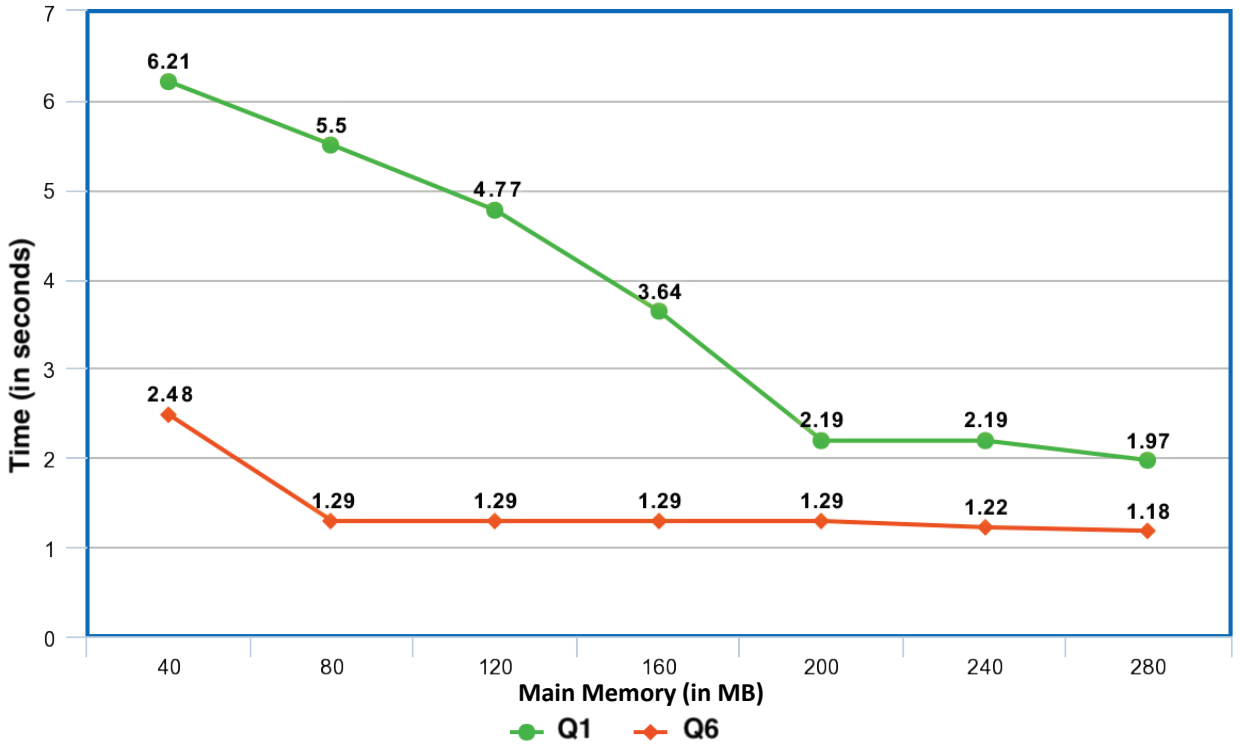


Q1 performs *slightly* worse due to cost of reconstruction

Q2 and Q3 perform 20% faster than baseline due to less number of irrelevant attributes

Experiment: TPC-H queries

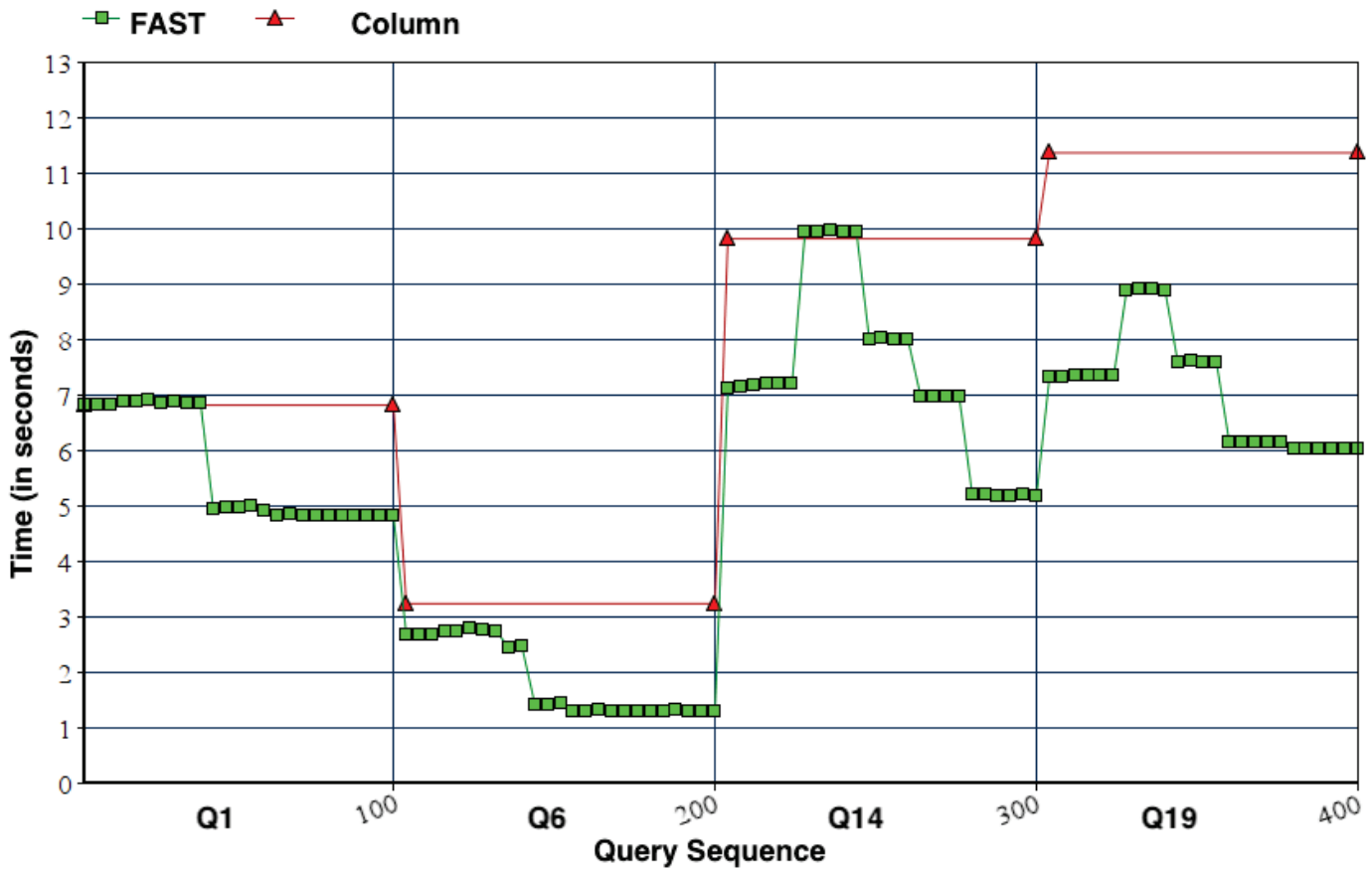




Experiment: Workload-aware adaptation

Sequence of TPC-H queries 1, 6, 14 and 19 - 100 queries of each type

Query	Fragments	Indexes
Q1	(*l_returnflag*, *l_linestatus*, *l_quantity*, *l_extendedprice*)	-
Q6	(*l_extendedprice*, *l_discount*)	-
Q14	(*l_partkey*), (*p_type*)	*p_partkey*
Q19	(*l_extendedprice*, *l_discount*)	*p_partkey*



Related work

- PAX – Ailamaki et al.
- Hyrise – Grund et al.
- Data Morphing – Hankins and Patel
- Flexible Storage Model – Arulraj et al.

Conclusion

- FAST provides sophisticated main memory buffer management
- Incorporates the best of row and column stores depending on the workload
- Performs better than traditional storage systems for real-world queries where only a fraction of attributes are relevant
- Dynamically adapts to an evolving workload without blocking query execution