Improving IBM Red Brick Warehouse Query Performance

Aman Sinha, Richard Taylor, Mandar Pimpale, David Wilhite, Cindy Fung

European Red Brick Users' Group Conference 2003

Milano, Italy **September 9 – September 10, 2003**

Talk Outline

- Red Brick Philosophy
- Basic Query Tuning Principles
  - More Tuning - Vista
- 6.3 Enhancements to Improve Query Performance:
  - Memory-mapping of dimension index/tables
  - Dynamic SmartScan optimization
  - Locally segmented TARGETjoin
  - TARGETjoin performance improvements
  - Optimizer hints to specify STARindex for joins
- Conclusions

Red Brick Philosophy

- Effective performance AND simple to use
- Apply intelligence on optimization internally
  - Minimize user intervention
  - A good example is Parallelism on Demand
    - A few simple guidelines to determine when and how much parallelism
- Conscientiously design very few tuning knobs
  - Designs focus on reducing complexity externally
  - Conduct studies to determine best default settings
    - Not necessary for customers to continuously tune
- Our goal is to provide the best performance with low cost of ownership
  - Performance enhancements adhere to this philosophy

Basic Query Tuning Principles

- Red Brick STARjoins are the fastest
- The most effective performance tuning is a good STAR schema
  - Will yield 80 – 99% performance gain
  - All other tuning efforts are relatively small refinements
- Mix of indexes (STARs, TARGETs, Btree's) are key to benefiting from STARjoin plans
  - Leading dimension constraints STARindex will perform best
  - Consider tradeoff with TARGET indexes to perform TARGETjoins when you can't create too many STARindexes
- Load fact table in STARindex order
  - Speeds up loads
  - Speeds up row fetches to the fact table
- Partition data/indexes effectively into PSUs and segments to maximize on parallelism speedup
  - See Query Performance Monitor case study

Basic Query Tuning Principles - 2

- Minimize spilling
  - Increase query memory limit, particularly for hashjoins
  - Same for optimized index builds, increase index tempspace
  - Use “set stats full;”, spilling reported in 1K values
    - User beware, not fully supported yet
  - Use several disks, or multi-disks logical volumes/disk arrays, for tempspace directories to minimize disk contention
    - Same for versioning logs – put on a disk array or logical volume
- Do not over commit on query parallelism
  - Join tasks per query should be equal or less than CPUs in system
  - Fetch tasks could be 1-2x join tasks, more if very slow I/O subsystem
- Do not be stingy with memory
  - 2-4 GB memory per CPU, more even better
  - Allows for more I/O caching in the OS file system cache
  - Allows for MMAPing for queries and loads

Basic Query Tuning Principles - 3

- Look at query execution statistics with "set stats info;"
  - Reports plan choice and index(es) selected, degrees of parallelism
  - Reports CPU and time (elapsed) of query
    - If elapsed time significantly greater than CPU time
      - Query is waiting for I/O
      - May require disk tuning or increase memory to cache data
    - If CPU time is too high
      - Investigate query plan improvement or create better index(es)
    - If there is parallelism, (CPU/time) should be greater than 1 to benefit
      - Linear speedup is ratio equals number of join tasks
- Optimize I/O performance
  - Separate indexes and data onto different disks, if not on a large striped volume
  - Configure enough memory to MMAP dimension indexes and tables
  - Minimum of 6-10 disks per CPU

Sample STARjoin Plans

- Single Fact STARjoin

RISQL> explain select sales.promokey, dollars
> from promotion, sales
> where sales.promokey = promotion.promokey
> and promotion.promo_type = 400;

EXPLANATION
[
- EXECUTE (ID: 0) 5 Table locks (table, type): (PROMOTION, Read_Only), (SALES, Read_Only), (PERIOD, Read_Only), (PRODUCT, Read_Only), (STORE, Read_Only)
--- CHOOSE PLAN (ID: 1) Num prelims: 1; Num choices: 3; Type: StarJoin;

Prelim: 1; Choose Plan [id : 1] {
BIT VECTOR SORT (ID: 2)
-- TABLE SCAN (ID: 3) Table: PROMOTION, Predicate: (PROMOTION.PROMO_TYPE) = (400) }

Sample STARjoin Plans - 2

- Single Fact STARjoin

Choice: 1; Choose Plan [id : 1] {
EXCHANGE (ID: 4) Exchange type: Functional Join
-- FUNCTIONAL JOIN (ID: 5) 1 tables: SALES
---- EXCHANGE (ID: 6) Exchange type: STARjoin
------ STARJOIN (ID: 7) Join type: InnerJoin, Num facts: 1,
Num potential dimensions: 4, Fact Table: SALES, Potential Indexes:
SALES_STAR_IDX, SALES_PROMO_STAR_IDX;
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION }

Choice: 2; Choose Plan [id : 1] {
EXCHANGE (ID: 8) Exchange type: Table Scan
-- FUNCTIONAL JOIN (ID: 9) 1 tables: PROMOTION
---- BTREE 1-1 MATCH (ID: 10) Join type: InnerJoin; Index(s):
[Table: PROMOTION, Index: PROMOTION_PK_IDX]
------ TABLE SCAN (ID: 11) Table: SALES, Predicate: <none> }

Choice: 3; Choose Plan [id : 1] {
EXCHANGE (ID: 12) Exchange type: Functional Join
-- FUNCTIONAL JOIN (ID: 13) 1 tables: SALES
---- EXCHANGE (ID: 14) Exchange type: TARGETjoin
------ TARGET JOIN (ID: 15) Table: SALES, Predicate: <none> ;
Num indexes: 1 Index(s): Index: PROMOKEY_TGTJOIN_IDX
-------- FUNCTIONAL JOIN (ID: 16) 1 tables: PROMOTION
---------- VIRTAB SCAN (ID: 17) }]

Sample STARjoin Plans - 3

- Fact-to-Fact STARjoin

```
RISQL> explain select week, store_name, prod_name,
> sum(sales.dollars) as sales,
> sum(sales_forecast.forecast_dollars) as forecast
> from period natural join sales natural join product natural join
> store natural join sales_forecast
> where year = 1998 and prod_name like ’Aroma%’
> group by week, store_name, prod_name
> having sales < forecast
> order by week, store_name, prod_name;
EXPLANATION
[
- EXECUTE (ID: 0) 6 Table locks (table, type): (PERIOD,
Read_Only), (PRODUCT, Read_Only), (STORE, Read_Only),
(SALES_FORECAST, Read_Only), (SALES, Read_Only), (PROMOTION,
Read_Only)

--- MERGE SORT (ID: 1) Distinct: FALSE
----- CHOOSE PLAN (ID: 2) Num prelims: 2; Num choices: 1; Type:
StarJoin;

Prelim: 1; Choose Plan [id : 2] {
BIT VECTOR SORT (ID: 3)
-- TABLE SCAN (ID: 4) Table: PERIOD, Predicate: (PERIOD.YEAR)
= (1998) }
```

Sample STARjoin Plans - 4

- Fact-to-Fact STARjoin

Prelim: 2; Choose Plan [id : 2] {
BIT VECTOR SORT (ID: 5)
-- TABLE SCAN (ID: 6) Table: PRODUCT, Predicate:
((PRODUCT.PROD_NAME) =< ('Aromaÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ')) &&
((PRODUCT.PROD_NAME) >= ('Aroma')) }

Choice: 1; Choose Plan [id : 2] {
HASH AVL AGGR (ID: 7) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
-- EXCHANGE (ID: 8) Exchange type: Functional Join
---- HASH AVL AGGR (ID: 9) Log Advisor Info: FALSE, Grouping:
TRUE, Distinct: FALSE;
------ FUNCTIONAL JOIN (ID: 10) 1 tables: PERIOD
-------- FUNCTIONAL JOIN (ID: 11) 1 tables: PRODUCT
---------- FUNCTIONAL JOIN (ID: 12) 1 tables: STORE
------------ FUNCTIONAL JOIN (ID: 13) 1 tables:
SALES_FORECAST
-------------- FUNCTIONAL JOIN (ID: 14) 1 tables: SALES
---------------- EXCHANGE (ID: 15) Exchange type: STARjoin
------------------ STARJOIN (ID: 16) Join type: InnerJoin,
Num facts: 2, Num potential dimensions: 4, Fact Table: SALES,
Potential Indexes: SALES_STAR_IDX, SALES_PROMO_STAR_IDX; Fact
Table: SALES_FORECAST, Potential Indexes: SALES_FORECAST_STAR_IDX;
Dimension Table(s): PERIOD, PRODUCT, STORE, PROMOTION }]

More Tuning - Vista Aggregates

- Materialize aggregates on precomputed views
- Excellent for significantly speeding up on aggregation queries
- Transparently rewrite queries to utilize pre-computed views
  - Step through all precomputed views for best aggregates to rewrite against
    - Rewrite against detail fact table with PK/FK relationships
    - Looks for functional dependencies if hierarchies are defined

      Example: *create hierarchy qtr_year*
      (from period(qtr) to period(year));
    - Single fact table only
- Use the vista advisor for suggestions on candidate precomputed views to create

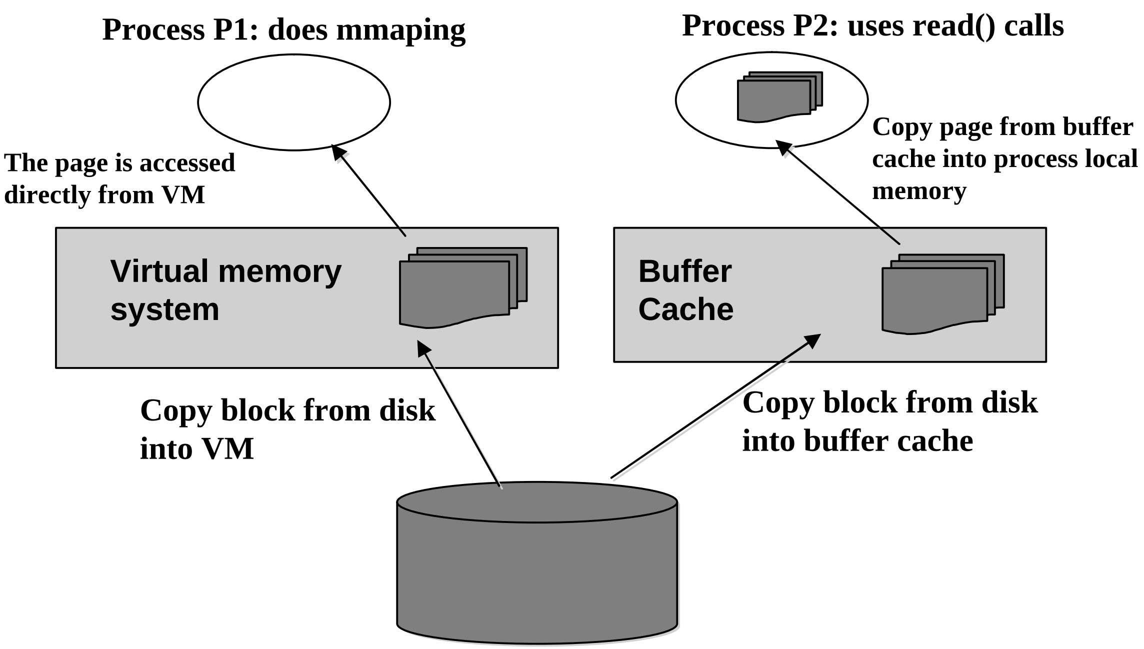
More Tuning - Vista and Automated Maintenance

- Requires more processing to maintain the aggregates up-to-date with any updates/deletes to the detail table
- Introduced automated aggregate maintenance in 6.1
  - Incremental or complete rebuild aggregates depending on amount of updates/deletes
  - Can be done automatically with loads (see TMU Tuning)
  - Strongly recommend adding count(*) to the aggregate table to facilitate incremental maintenance
- Continually improving vista rewrite capability and maintenance
  - Extend incremental maintenance on nullable columns in 6.3

6.3 Enhancements to Improve Query Performance

- Memory-mapping of dimension index/tables
- Dynamic SmartScan optimization
- Locally segmented TARGETjoin
- TARGETjoin performance improvements
- Optimizer hints to specify STARindex for joins

Memory-mapped I/O Overview



Memory-mapping in Red Brick Server

- Server in 6.3 performs memory-mapping of dimension tables and indexes for selected operators
  - Applies to StarJoin/TargetJoin/TableScan plans that contain *Btree-1-1-Match* or *FunctionalJoin* operators
  - Improves cache locality especially for large dimensions
    - Potential to significantly reduce number of *read()* system calls – thus reduced cpu and I/O overhead
  - Maintains a single shared read-only copy of dimension data for concurrent queries
  - Makes intelligent decisions about mmap resource allocation among operators
  - Prioritizes among tables and indexes of different sizes
  - Provides good speedup (from 5 to 150% seen in certain queries)

MMAP External Interface

- SET QUERY MMAP {ON | OFF} [Default = ON]
  - Can be set per-session or across all sessions using a config file option

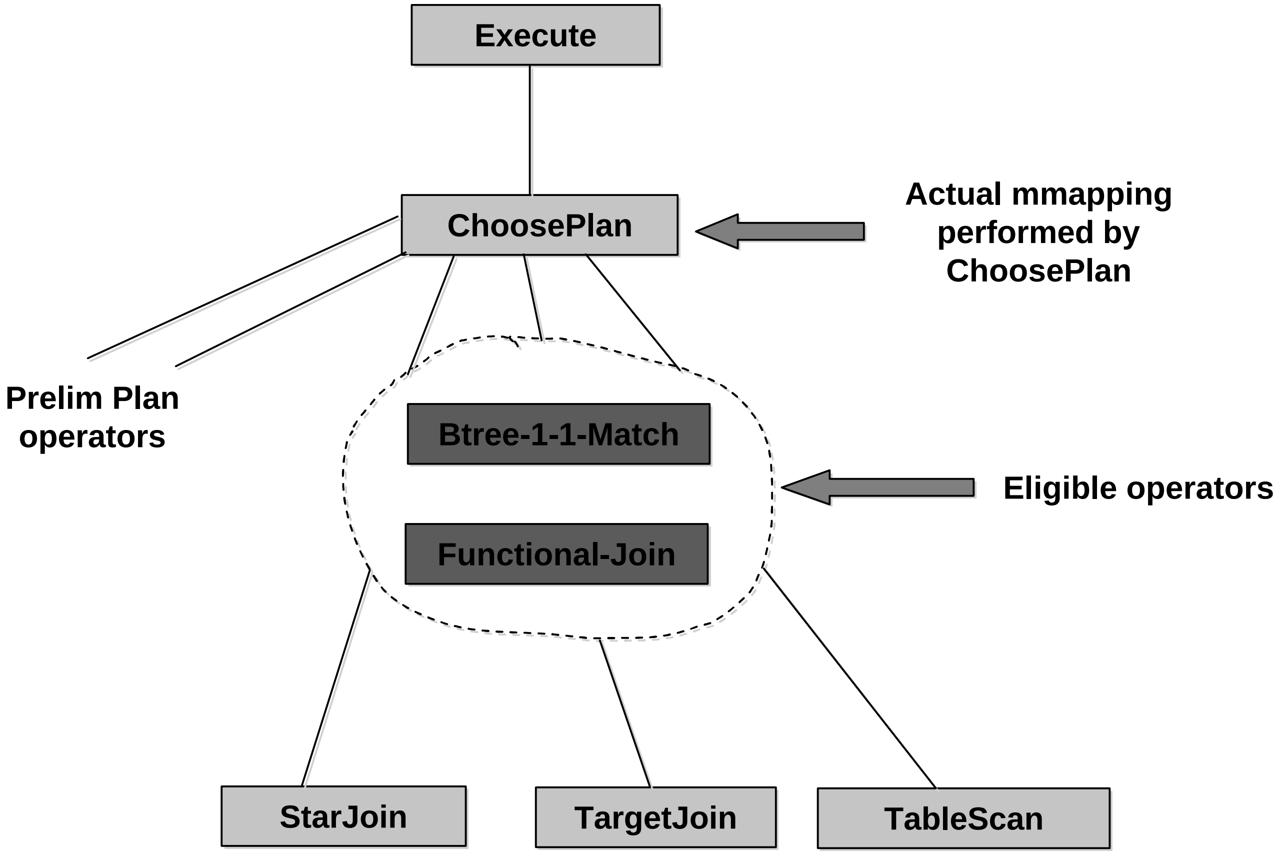
- SET QUERY MMAP LIMIT *value* {K|M|G} [Default = 5MB]
  - Similar to Query Memory Limit …but not limited to 2GB
    - From 8KB up to ULONG_MAX (several thousand Terabytes on 64 bit platforms)

- Example messages:
  - ** INFORMATION ** (9151) CHOOSE PLAN (ID: 1) Index DIM01_PK_IDX of table DIM01 is **100.00** percent memory-mapped**.**
  - ** INFORMATION ** (9153) CHOOSE PLAN (ID: 1) Table DIM01 is **45.00** percent memory-mapped**.**
- Statistics:
  - MMAP_READS and CUM_MMAP_READS columns in the DST_PERFORMANCE_OPSTATS table

MMAP Memory

- Mmap memory serves as a shared cache of dimension data among multiple concurrent queries
  - However, different queries can set different mmap memory limit
- In addition to the Red Brick block cache…
  - …however, reads to mmapped data go directly to the mmap space, not redirected from the block cache
  - User may see fewer block cache hits but at the same time higher mmap space hits
- Mmap memory and Query memory compete for the same physical memory
  - Important to consider this when increasing the mmap memory

Operators Eligible for MMAPing



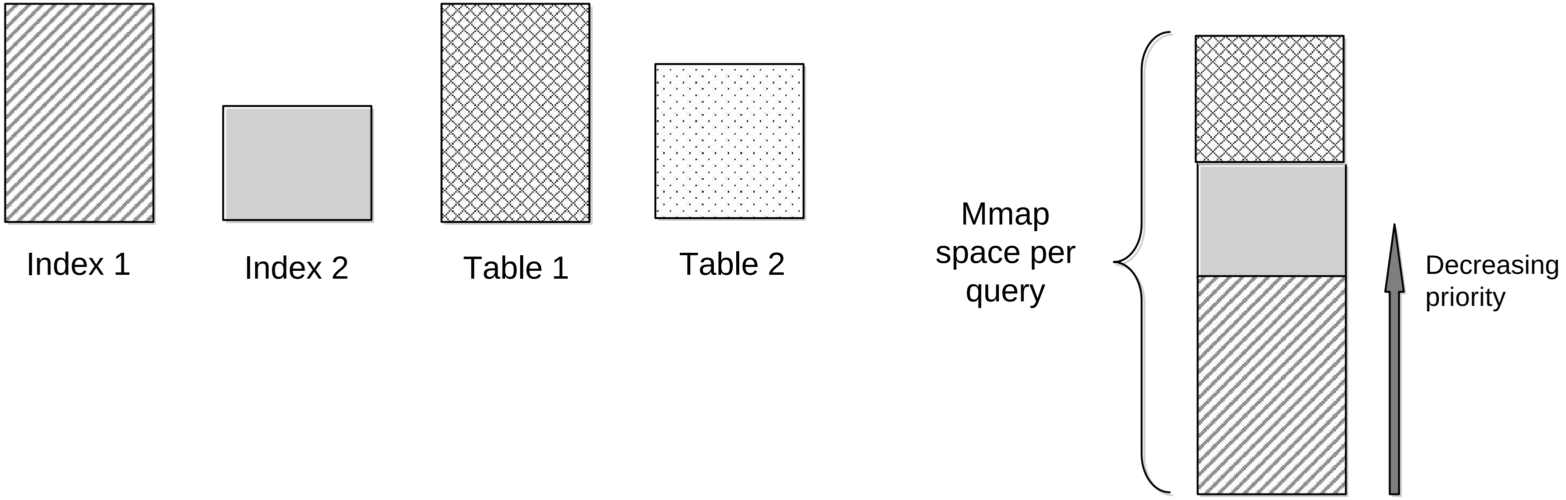
Criteria for MMAPing

- Only StarJoin type ChoosePlan operator performs mmapping of tables and indexes
  - Multiple ChoosePlan operators in a query plan will share the mmap memory resources
  - Resource allocation gives higher preference to ChoosePlans that are higher in the query plan hierarchy
- Only B11Ms and FJs in Choice plans are eligible
- B11M must be to a dimension Primary-key index
- FJ must be to a dimension table, not fact
- Leading dimension table of star-index is not mmapped
- Mmapping is not performed if fewer than 1000 rows are selected from the fact table

MMAP Priority Among Tables and Indexes

- Priority based on Object type (table/index) and Size
  - Index given higher priority compared to table
  - Larger objects given higher priority

- Example:



Example Query and its (Partial) Explain

```
SELECT  city_name, customer_name,
     sum(num_orders)
FROM  sales s, city c, customer cu
WHERE s.city_id = c.city_id
AND s.customer_id = cu.customer_id
 AND city_name LIKE  'Los%'
AND customer_name LIKE 'Joe%'
GROUP BY city_name,
     customer_name;
```

# of rows: Sales: 5 million, City: 230, Customer: 1 million
Query memory limit: 50MB, Parallelism: 3

- EXECUTE (ID: 0)

--- CHOOSE PLAN (ID: 1) Num prelims: 1; Num choices: 2; Type: StarJoin;

Prelim: 1; Choose Plan [id : 1] {

BIT VECTOR SORT (ID: 2)

-- TABLE SCAN (ID: 3) Table: CY (CITY), Predicate: ((CY.CITY_NAME) =< ('Hÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ')) && ((CY.CITY_NAME) >= ('H'))

}

Choice: 1; Choose Plan [id : 1] {

HASH AVL AGGR (ID: 4);

-- EXCHANGE (ID: 5) Exchange type: Functional Join

---- HASH AVL AGGR (ID: 6)

------ FUNCTIONAL JOIN (ID: 7) 1 tables: CU (CUSTOMER)

Mmap eligible {

-------- BTREE 1-1 MATCH (ID: 8) Join type: InnerJoin; Index(s): [Table: CUSTOMER, Index: CUSTOMER_PK_IDX]

---------- FUNCTIONAL JOIN (ID: 9) 1 tables: CY (CITY)

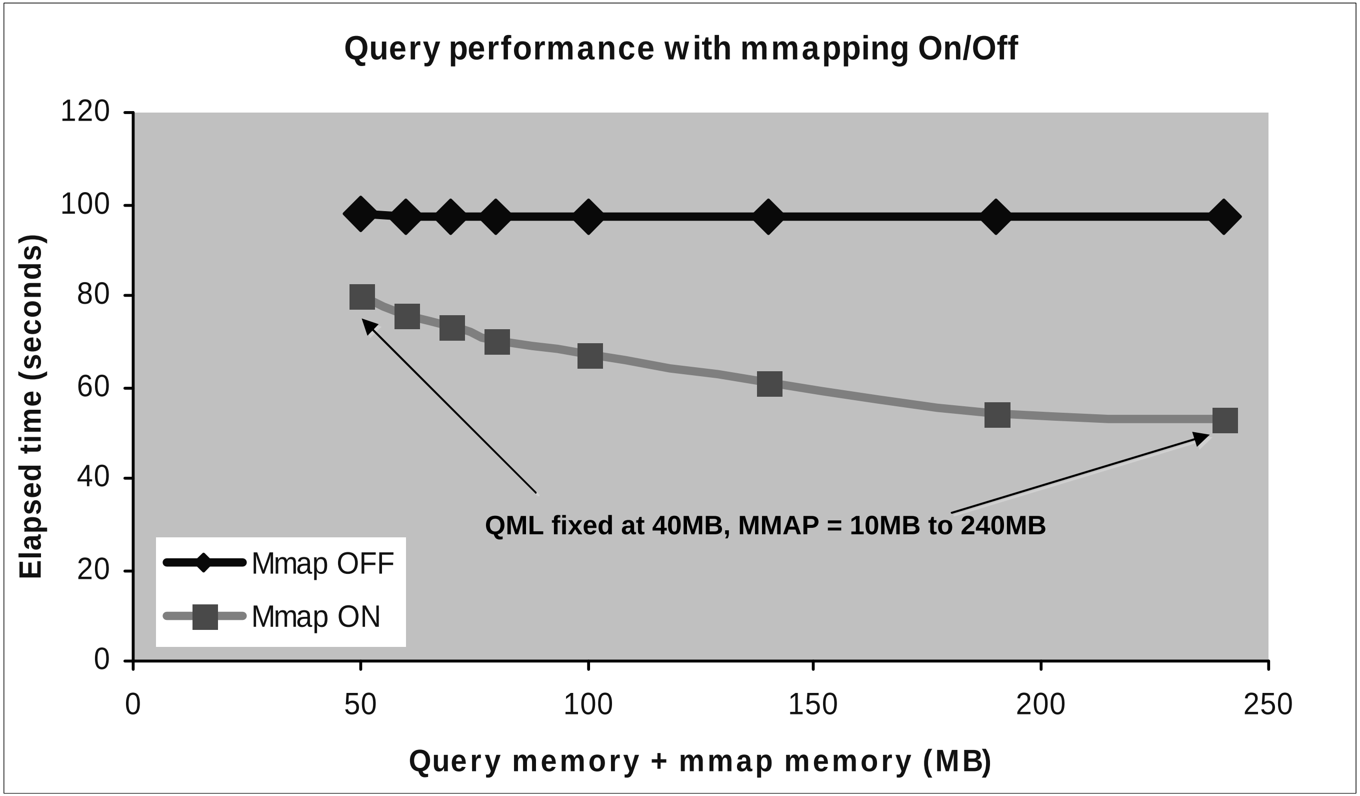
}

------------ FUNCTIONAL JOIN (ID: 10) 1 tables: S (SALES)

-------------- EXCHANGE (ID: 11) Exchange type: STARjoin

---------------- STARJOIN (ID: 12) Join type: InnerJoin, Num facts: 1, Num potential dimensions: 4, Fact Table: S (SALES), Potential Indexes: SALES_STAR1 ; Dimension Table(s): CY (CITY), PROD, MFR, PERIOD

MMAP Performance



- 10MB – 200MB of mmap memory gave 20% - 100% performance improvement

MMAP Performance