query
processing
Query Processing

- The query processor turns user queries and data modification commands into a query plan - a sequence of operations (or algorithm) on the database from high level queries to low level commands.

- **Decisions taken by the query processor**
  - Which of the algebraically equivalent forms of a query will lead to the most efficient algorithm?
  - For each algebraic operator what algorithm should we use to run the operator?
  - How should the operators pass data from one to the other? (eg, main memory buffers, disk buffers)
Query Processing Example

SQL Query

```sql
SELECT B,D
FROM R,S
WHERE R.A = 'c' AND S.E = 2 AND R.C=S.C
```

Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>z</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>y</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>x</td>
</tr>
</tbody>
</table>
How do we execute the query eventually?

- Scan relations
- Compute Cartesian product
- Select tuples
- Compute projection

One idea!

R × S

<table>
<thead>
<tr>
<th>R.A</th>
<th>R.B</th>
<th>R.C</th>
<th>S.C</th>
<th>S.D</th>
<th>S.E</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

We found one!
Describing execution plans
Using an enhanced version of the relational algebra

PLAN I

\[ \Pi_{B,D} \sigma_{R.A = "c" \land S.E = 2 \land R.C = S.C} (R \times S) \]

1. Scan R
2. For each tuple r of R scan S
3. For each (r,s), where s in S
   select and project on the fly

or:

\[ \Pi_{B,D} [ \sigma_{R.A = "c" \land S.E = 2 \land R.C = S.C} (R \times S) ] \]
Describing execution plans
Using an enhanced version of the relational algebra

PLAN I

\[ \Pi_{B,D} \quad \sigma_{R.A=\text{"C"} \land S.E=2 \land R.C=S.C} \quad X \]

1. Scan R
2. For each tuple r of R scan S
3. For each \((r,s)\), where s in S select and project on the fly

FLY & SCAN are the defaults!
Describing execution plans
Using an enhanced version of the relational algebra

PLAN II

\[
\begin{align*}
\Pi_{B,D} & \quad \text{HASH} \\
\sigma_{R.A = "c"} & \quad \sigma_{S.E = 2} \\
R & \quad S
\end{align*}
\]

1. Scan R & S
2. Perform on the fly selections
3. Do hash join
4. Project

\[\bowtie\quad \text{natural join}\]
Describing execution plans
Using an enhanced version of the relational algebra
Describing execution plans
Using an enhanced version of the relational algebra

PLAN III

\[ \Pi_{R.B, S.D} \sigma_{S.E=2} \]

Use R.A and S.C Indexes
1. Use R.A index to select R tuples with R.A = “c”
2. For each R.C value found, use S.C index to find matching join tuples
3. Eliminate join tuples S.E \neq 2
4. Project B,D attributes

\( \bowtie^{RI} \): right index natural join
Describing execution plans
Using an enhanced version of the relational algebra

PLAN III

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>3</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>y</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>z</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>y</td>
<td>3</td>
</tr>
</tbody>
</table>

A = “c”

next tuple: <c,7,15>

output: <2,x>

check = 2?

I1 → I2
From Query To Optimal Plan

• Complex process
• Algebra-based logical and physical plans
• Transformations
• Evaluation of multiple alternatives
Issues in Query Processing and Optimization

• Generate Plans
  Employ efficient execution primitives for computing relational algebra operations
  Systematically transform expressions to achieve more efficient combinations of operators

• Estimate Cost of Generated Plans
  Statistics

• “Smart” Search of the Space of Possible Plans
  Always do the “good” transformations (relational algebra optimization)
  Prune the space (e.g., System R)

• Often the above steps are mixed
The Journey of a Query

1. SQL Query
2. Parse
   - Parse tree
3. Convert
   - Logical query plan
4. Generate/transform lqp's
5. Estimate result sizes
   - "Improved" logical query plans
6. Generate physical plans
7. Execute
   - Pick best
8. Estimate costs
9. Generate/transform pqp's

Scope of responsibility of each module is fuzzy
The Journey of a Query

Logical query plan generator

SELECT Theater
FROM Movie M, Schedule S
WHERE
  M.Title = S.Title
AND M.Actor = "Winger"

Parse/Convert

Initial logical query plan
The Journey of a Query

Logical Plan Generator applies Algebraic Rewriting

Another logical query plan

Another logical query plan
Logical query plan generator

Logical Plan Generator applies Algebraic Rewriting

Another logical query plan

Summary of Logical Plan Generator:
- 4 logical query plans created
- Algebraic rewritings were used for producing the candidate logical query plans
- The last one is the winner (at least, cannot be a big loser)
- In general, multiple logical plans may “win” eventually
The Journey of a Query

Physical query plan generator

1st physical query plan

\[ \pi_{\text{Theater}} \]

\[ \sigma_{\text{Actor} = \text{"Winger"}} \]

\[ \text{LEFT INDEX} \]

S.Title = M.Title

Index on Actor and Title, unsorted tables, Table size >> Memory size

Physical Plan Generator chooses execution primitives and data passing
Physical query plan generator

The Journey of a Query

Physical Plan Generator chooses execution primitives and data passing

2nd physical query plan

\[ \pi_{\text{Theater}} \]
\[ \text{SORT-MERGE} \]
\[ S.\text{Title}=M.\text{Title} \]
\[ \sigma \]
\[ \text{INDEX} \]
\[ \text{Actor}=\text{“Winger”} \]

Index on Actor, Table Schedule sorted on Schedule

18
The Journey of a Query

Physical query plan generator

Summary of Physical Plan Generator:

- 2 logical query plans created in this case.
- In general, more than one different logical plans may be generated by choosing different primitives.
Example: Nested SQL Query

```
SELECT title
FROM StarsIn
WHERE starName IN (  
    SELECT name  
    FROM MovieStar  
    WHERE birthdate LIKE ‘%1960’)

(Find the movies with stars born in 1960)
```
Example: Parse Tree

```
SELECT <SelList>
FROM <FromList>
WHERE <Condition>
    <Tuple> IN <Query>
    
SELECT      <SelList>    FROM     <FromList>     WHERE     <Condition>
    <Attribute>           <RelName>         <Attribute>  LIKE  <Pattern>

Example:
SELECT name FROM MovieStar WHERE birthDate LIKE '%1960'
    (`%1960`)
```
Example: Generating Rel. Algebra

\[ \Pi_{\text{title}} \sigma \text{StarsIn} \langle \text{condition} \rangle \langle \text{tuple} \rangle \text{IN} \Pi_{\text{Name}} \langle \text{attribute} \rangle \sigma \text{birthdate LIKE ‘%1960’} \langle \text{starName} \rangle \text{MovieStar} \]

An expression using a two-argument \( \sigma \), midway between a parse tree and relational algebra
Example: Logical Query Plan

\[ \Pi_{\text{title}} \left( \sigma_{\text{starName}=\text{name}} \left( \times \left( \Pi_{\text{name}} \left( \sigma_{\text{birthdate LIKE ‘%1960’}} \left( \text{StarsIn} \right) \right) \right) \right) \right) \]

May consider “IN” elimination as a rewriting in the logical plan generator or may consider it a task of the converter.
Example: Improved Logical Plan

\[ \Pi_{\text{title}} \]

\[ \sigma_{\text{birthdate LIKE } '1960'} \]

\[ \Pi_{\text{name}} \]

\[ \Pi_{\text{name}} \]

\[ \text{MovieStar} \]

\[ \text{StarsIn} \]

\[ \text{starName = name} \]
Example: Size estimation

Sizes are important for selecting physical query plans

- StarsIn
  - Need expected size

\[ \Pi \sigma \text{MovieStar} \]
Example: One Physical Plan

\[ \Pi_{\text{title}} \]
\[ \sigma \text{birthdate LIKE } '%1960' \]
\[ \Pi_{\text{name}} \]
\[ \text{INDEX starName = name} \]
\[ \text{Π}_{\text{name}} \]
\[ \text{Π}_{\text{title}} \]

Additional parameters:
memory size, result sizes...
Relational Algebra Optimization

- Transformation rules (preserve equivalence)
- What are good transformations?
Algebraic Rewritings

Commutativity & Associativity

**Cartesian Product**

Commutativity:

\[ R \times S \rightarrow S \times R \]

**Natural Join**

Commutativity:

\[ R \bowtie S \rightarrow S \bowtie R \]

**Associativity**

\[ R \times (S \times T) \rightarrow (R \times S) \times T \]

\[ R \bowtie (S \bowtie T) \rightarrow (R \bowtie S) \bowtie T \]
Algebraic Rewritings

Commutativity & Associativity

Commutativity

Union

Intersection

Associativity
Algebraic Rewritings
Decomposition of Logical Connectives
Decomposition of Negation

\[ \sigma_{\text{cond1 AND NOT cond2}} R \leftrightarrow \sigma_{\text{NOT cond2}} R \]
Algebraic Rewritings
Pushing Selection through
Union & Difference

Union:
\[ \sigma_{\text{cond}} R \cup \sigma_{\text{cond}} S \]

Difference:
\[ \sigma_{\text{cond}} R \setminus \sigma_{\text{cond}} S \]

What about intersection?
Algebraic Rewritings
Pushing Selection through Cartesian Product & Join

The right direction requires that \( \text{cond} \) refers to \( S \) attributes only.

The right direction requires that \( \text{cond} \) refers to \( S \) attributes only.

The right direction requires that all the attributes in \( \text{cond} \) appear in both \( R \) and \( S \).
Rules: \( \pi, \sigma \) combined

Let \( x \) = subset of \( R \) attributes

\( z \) = attributes in predicate \( P \) (subset of \( R \) attributes)

\[
\pi_x[\sigma_p(R)] = \pi_x \{ \sigma_p [ \pi_x(R) ] \}
\]
Algebraic Rewritings

Projection Decomposition

\[
\begin{array}{c|c}
\pi_X & \pi_{XY} \\
X & R \\
\end{array}
\]

\[
\begin{array}{c|c}
\pi_X & \pi_X \\
X & R \\
\end{array}
\]
Derived Rules: $\sigma + \text{combined}$

More Rules can be Derived:

$\sigma p \land q (R \bowtie S) = ?$

where $p$ only at $R$, $q$ only at $S$
Derived Rules: $\sigma + \bowtie$ combined

More Rules can be Derived:

\[ \sigma_{p \land q} (R \bowtie S) = \]
\[ \sigma_p [ \sigma_q (R \bowtie S) ] = \]
\[ \sigma_p [ R \bowtie \sigma_q (S) ] = \]
\[ [ \sigma_p (R)] \bowtie [\sigma_q (S)] \]

where $p$ only at $R$, $q$ only at $S$
Which are always “good” transformations?

- \( \sigma_{p_1 \land p_2} (R) \rightarrow \sigma_{p_1} [\sigma_{p_2} (R)] \)
- \( \sigma_p (R \bowtie S) \rightarrow [\sigma_p (R)] \bowtie S \)
- \( R \bowtie S \rightarrow S \bowtie R \)
- \( \pi_x [\sigma_p (R)] \rightarrow \pi_x \{\sigma_p [\pi_{xz} (R)]\} \)
Bottom Line

• No transformation is always good at the l.q.p level
• Usually good:
  early selections
  elimination of cartesian products
  elimination of redundant subexpressions
• Many transformations lead to “promising” plans
  Commuting/rearranging joins
  In practice too “combinatorially explosive” to be handled as rewriting of l.q.p.
Arranging the Join Order: The Wong-Youssefi algorithm (INGRES)

Sample TPC-H Schema
Nation(NationKey, NName)
Customer(CustKey, CName, NationKey)
Order(OrderKey, CustKey, Status)
LineItem(OrderKey, PartKey, Quantity)
Product(SuppKey, PartKey, PName)
Supplier(SuppKey, Sname)

SELECT SName
FROM Nation, Customer, Order, LineItem, Product, Supplier
WHERE Nation.NationKey = Customer.NationKey
    AND Customer.CustKey = Order.CustKey
    AND Order.OrderKey = LineItem.OrderKey
    AND LineItem.PartKey = Product.PartKey
    AND Product.Suppkey = Supplier.SuppKey
    AND NName = “Canada”

Find the names of suppliers that sell a product that appears in a line item of an order made by a customer who is in Canada.
Challenges with Large Natural Join Expressions

For simplicity, assume that in the query
1. All joins are natural
2. whenever two tables of the FROM clause have common attributes we join on them
3. Consider Right-Index only

One possible order
Multiple Possible Orders

Order → Lineltem → Product → Supplier

π_{SName}

σ_{NName="Canada"}

Customer → Nation
Wong-Yussefi algorithm assumptions and objectives

- Assumption 1 (weak): Indexes on all join attributes (keys and foreign keys)
- Assumption 2 (strong): At least one selection creates a *small* relation
  
  A join with a small relation results in a small relation

- Objective: Create sequence of index-based joins such that all intermediate results are small
Hypergraphs

- relation hyperedges
  - two hyperedges for same relation are possible
- each node is an attribute
- can extend for non-natural equality joins by merging nodes
Small Relations/Hypergraph Reduction

“Nation” is small because it has the equality selection \[ \sigma_{\text{NName} = “Canada”} \] to start the plan.

Pick a small relation (and its conditions) to start the plan.
Remove small relation (hypergraph reduction) and color as “small” any relation that joins with the removed “small” relation.

Pick a small relation (and its conditions if any) and join it with the small relation that has been reduced.
After a bunch of steps...

\[ \sigma_{\text{NName} = "Canada"} \]

\[ \pi_{\text{SName}} \]
SELECT S.SName
FROM Nation, Customer, Order, Lineltem L, Product P, Supplier S,
    Lineltem LE, Product PE, Supplier Enron
WHERE Nation.NationKey = Customer.NationKey
    AND Customer.CustKey = Order.CustKey
    AND Order.OrderKey = L.OrderKey
    AND L.PartKey = P.Partkey
    AND P.Suppkey = S.SuppKey
    AND Order.OrderKey = LE.OrderKey
    AND LE.PartKey = PE.Partkey
    AND PE.Suppkey = Enron.SuppKey
    AND Enron.Sname = "Enron"
    AND NName = "Cayman"
Multiple Instances of Each Relation
Multiple choices are possible
Algorithms for Relational Algebra Operators

• Three primary techniques
  Sorting
  Hashing
  Indexing

• Three degrees of difficulty
  Data small enough to fit in memory
  Too large to fit in main memory but small enough to be handled by a “two-pass” algorithm
  So large that “two-pass” methods have to be generalized to “multi-pass” methods (quite unlikely nowadays)
Join Implementations

- Nested-loop Join
- Merge Join
- Index Join
- Hash Join
Join Example

R1 $\bowtie$ R2 on common attribute C
Nested-Loop Join

- Nested loop join
  (conceptually, without taking into account disk block issues)

  for each $r \in R_1$ do
    for each $s \in R_2$ do
      if $r.C = s.C$ then output $r,s$ pair
Merge Join

- Merge join (*conceptually*)
  1. if R1 and R2 not sorted, sort them
  2. $i \leftarrow 1; j \leftarrow 1;$
  
  While $(i \leq T(R1)) \land (j \leq T(R2))$ do
  
  - if $R1\{i\}.C = R2\{j\}.C$ then outputTuples
  - else if $R1\{i\}.C > R2\{j\}.C$ then $j \leftarrow j+1$
  - else if $R1\{i\}.C < R2\{j\}.C$ then $i \leftarrow i+1$
Merge Join (continued)

Procedure Output-Tuples

While (R1{i}.C = R2{j}.C) \land (i \leq T(R1)) do

[jj ← j;
while (R1{i}.C = R2{jj}.C) \land (jj \leq T(R2)) do

[output pair R1{i}, R2{jj};

jj ← jj+1 ]

i ← i+1 ]
## Merge Join Example

<table>
<thead>
<tr>
<th>i</th>
<th>R1{i}.C</th>
<th>R2{j}.C</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>40</td>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>52</td>
<td>7</td>
</tr>
</tbody>
</table>
Index Join

- Index Join (conceptually)

For each \( r \in R_1 \) do

\[ X \leftarrow \text{index (} R_2, C, r.C \text{)} \]

for each \( s \in X \) do

output \( r,s \) pair

Assume \( R_2.C \) index

Note: \( X \leftarrow \text{index} \text{(rel, attr, value)} \)
means that \( X = \text{set of rel tuples with attr = value} \)
Hash Join

- Hash Join (conceptually)
  1. Hash R1 tuples into G buckets
  2. Hash R2 tuples into H buckets
  3. For \( i = 0 \) to \( k \) do
     - match tuples in \( G_i \), \( H_i \) buckets

Hash function \( h \), range \( 0 \rightarrow k \)
Buckets for R1: \( G_0, G_1, \ldots, G_k \)
Buckets for R2: \( H_0, H_1, \ldots, H_k \)
Hashing Example: Even/Odd

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
<th>Buckets</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>Even</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>Odd</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td></td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
<td></td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
<td></td>
<td>11</td>
<td>14</td>
</tr>
</tbody>
</table>

Buckets: Even: 2 4 8; Odd: 3 5 9
Factors affecting performance

(1) Tuples of relation stored physically together?
(2) Relations sorted by join attribute?
(3) Indexes exist?
More details

• The above algorithms are conceptual and do not account for physical implementation details.
• In reality we have to specify exactly how the algorithm will operate in terms of disk/memory accesses.
• Then based on this information we can calculate the cost of each operator in terms of disk accesses.
Disk-oriented Computation Model

- There are $M$ main memory buffers. Each buffer has the size of a disk block.
- The input relation is read one block at a time.
- The cost is the number of blocks read.
- If $B$ consecutive blocks are read the cost is $B/d$.
- The output buffers are not part of the $M$ buffers mentioned above.

Pipelining allows the output buffers of an operator to be the input of the next one.

We do not count the cost of writing the output.
Disk-oriented Computation Model

Metric: # of I/Os (ignoring writing the result)

Caution: This may not be the best way to compare
• ignoring CPU costs
• ignoring timing
• ignoring double buffering requirements
Notation

- \( M = \# \) memory blocks available
- \( B(R) = \) number of blocks that \( R \) occupies
- \( T(R) = \) number of tuples of \( R \)
- \( V(R,[a_1, a_2, \ldots, a_n]) = \) number of distinct tuples in the projection of \( R \) on \( a_1, a_2, \ldots, a_n \)
One-Pass Main Memory Algorithms for Unary Operators

• Assumption: Enough memory to keep the relation
• Projection and selection:
  Scan the input relation $R$ and apply operator one tuple at a time
  Incremental cost of “on the fly” operators is 0
  Cost depends on
  • clustering of $R$
  • whether the blocks are consecutive
• Duplicate elimination and aggregation
  Create one entry for each group and compute the aggregated value of the group
  It becomes hard to assume that CPU cost is negligible
  • main memory data structures are needed
One-Pass Nested Loop Join

- Assume \( B(R) \) is less than \( M \)
- Tuples of \( R \) should be stored in an efficient lookup structure
- Algorithm:
  
  ```
  for each block \( B_r \) of \( R \) do 
  store tuples of \( B_r \) in main memory 
  for each each block \( B_s \) of \( S \) do 
  for each tuple \( s \) of \( B_s \) 
  join tuples of \( s \) with matching tuples of \( R \) 
  ```

What is the cost of this algorithm?
Generalization of Nested-Loops

- Works even if \( B(R) \) is greater than \( M \)

for each chunk of \( M-1 \) blocks \( Br \) of \( R \) do
  store tuples of \( Br \) in main memory
  for each each block \( Bs \) of \( S \) do
    for each tuple \( s \) of \( Bs \)
      join tuples of \( s \) with matching tuples of \( R \)

What is the cost of this algorithm?
Two-Pass Hash-Based Algorithms

• General Idea: Hash the tuples of the input arguments in such a way that all tuples that must be considered together will have hashed to the same hash value. If there are $M$ buffers pick $M-1$ as the number of hash buckets.

• Example: Duplicate Elimination
  Phase 1: Hash each tuple of each input block into one of the $M-1$ bucket/buffers. When a buffer fills save to disk.
  Phase 2: For each bucket:
    • load the bucket in main memory,
    • treat the bucket as a small relation and eliminate duplicates
    • save the bucket back to disk.

Catch: Each bucket has to be less than $M$.
Cost = ?