CSE 544: Lecture 13

Monday, 5/13/2002

Hash table indexes, query optimization

Outline

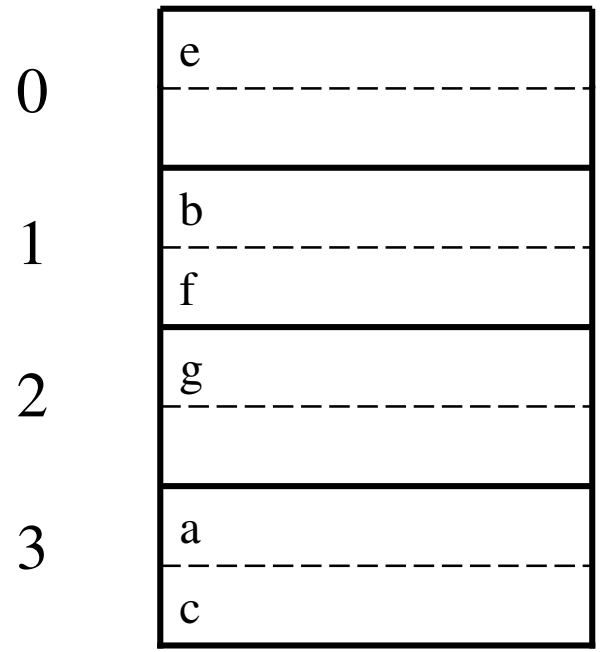
- Hash tables: chapter 10
- Query optimization:
  - chapters 13, 14
  - Chaudhuri, *An overview of query optimization in relational systems*

Hash Tables

- Secondary storage hash tables are much like main memory ones
- Recall basics:
  - There are n *buckets*
  - A hash function f(k) maps a key k to {0, 1, …, n-1}
  - Store in bucket f(k) a pointer to record with key k
- Secondary storage: bucket = block, use overflow blocks when needed

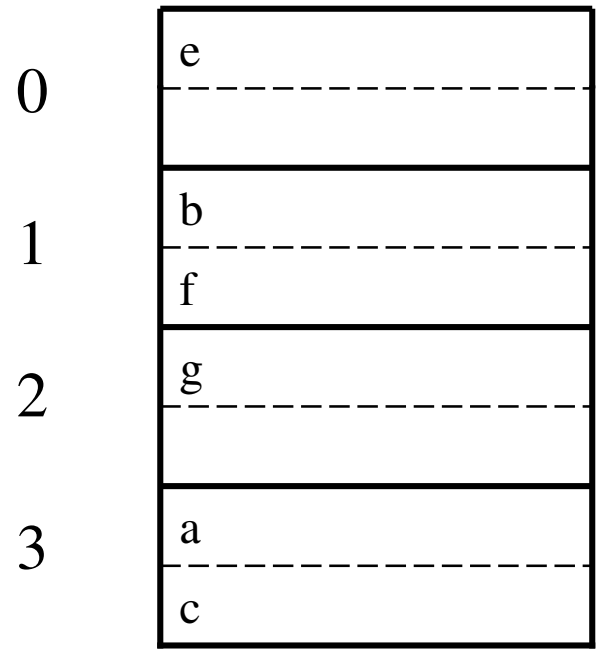
Hash Table Example

- Assume 1 bucket (block) stores 2 keys + pointers
- h(e)=0
- h(b)=h(f)=1
- h(g)=2
- h(a)=h(c)=3



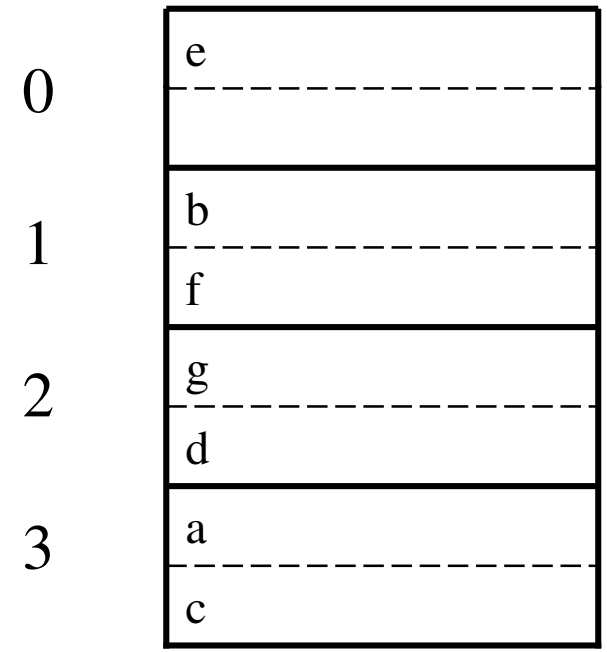
Searching in a Hash Table

- Search for a:
- Compute h(a)=3
- Read bucket 3
- 1 disk access



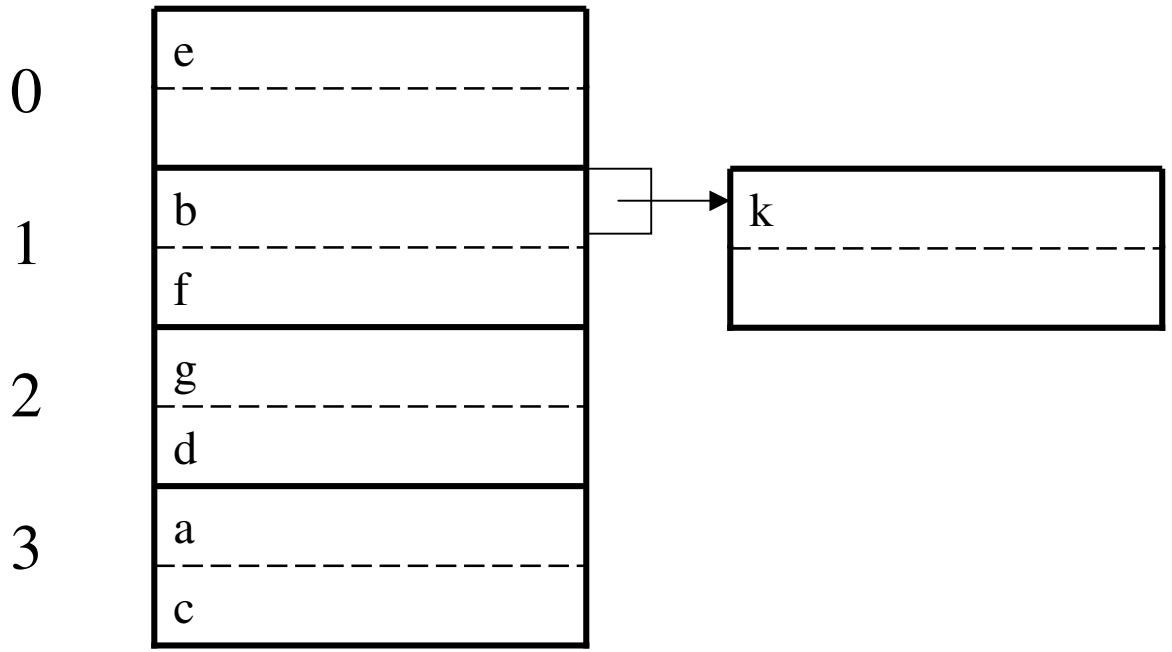
Insertion in Hash Table

- Place in right bucket, if space
- E.g. h(d)=2



Insertion in Hash Table

- Create overflow block, if no space
- E.g. h(k)=1



- More over-flow blocks may be needed

Hash Table Performance

- Excellent, if no overflow blocks
- Degrades considerably when number of keys exceeds the number of buckets (I.e. many overflow blocks).

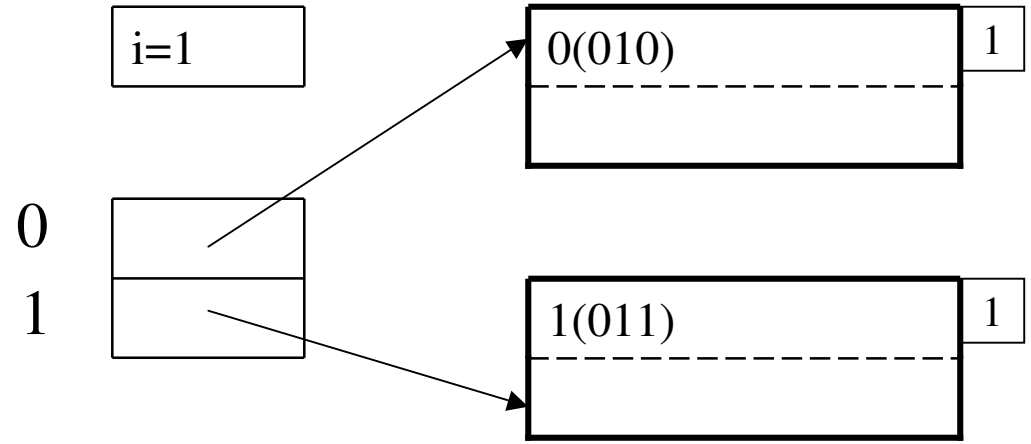
Extensible Hash Table

- Allows has table to grow, to avoid performance degradation
- Assume a hash function h that returns numbers in $\{0, \ldots, 2^k - 1\}$
- Start with $n = 2^i << 2^k$, only look at first i most significant bits

Remark: textbook looks at least significant bits first (no big deal)

Extensible Hash Table

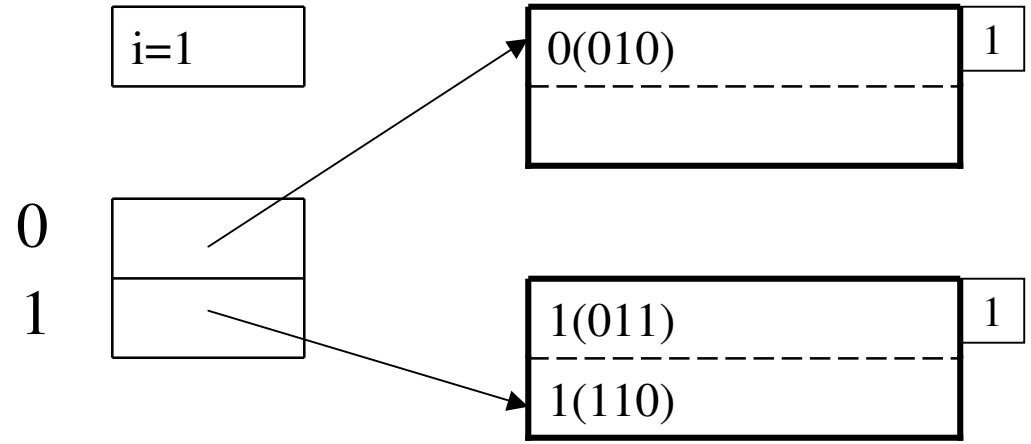
- E.g. i=1, n=2, k=4



- Note: we only look at the first bit (0 or 1)

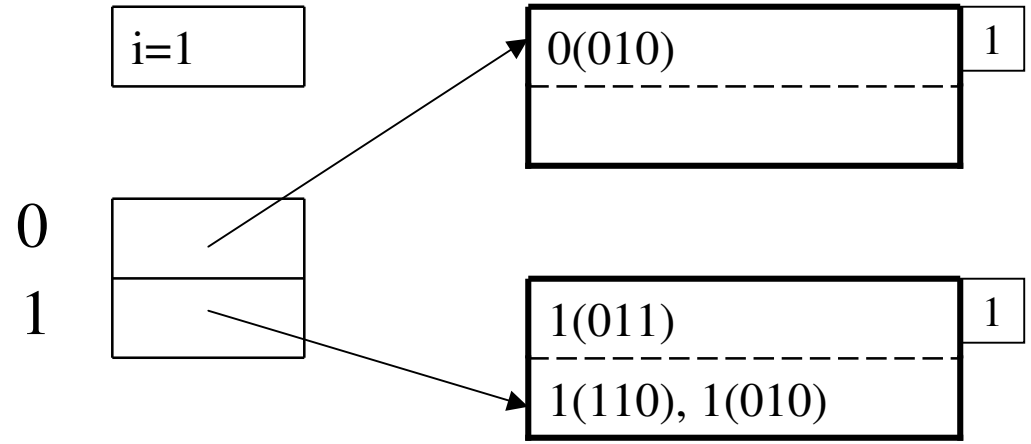
Insertion in Extensible Hash Table

- Insert 1110



Insertion in Extensible Hash Table

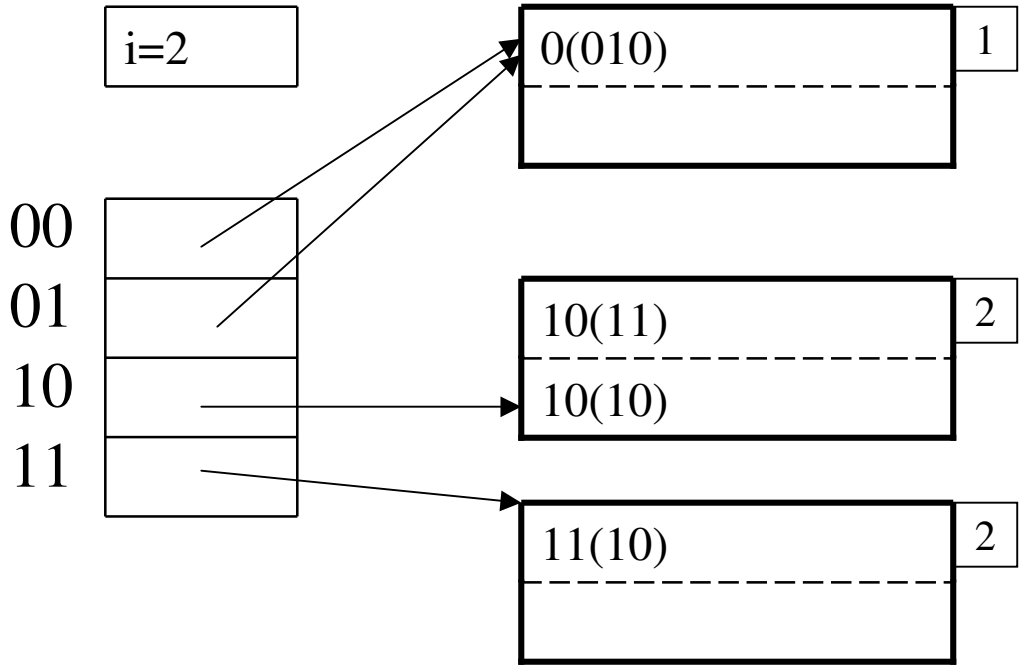
- Now insert 1010



- Need to extend table, split blocks
- i becomes 2

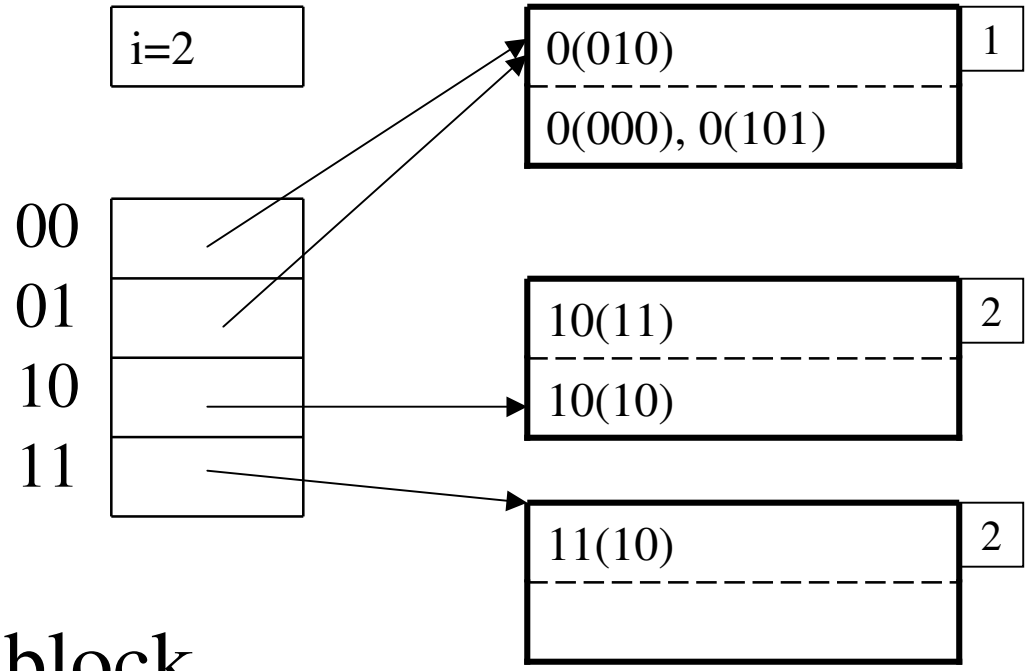
Insertion in Extensible Hash Table

- Now insert 1110



Insertion in Extensible Hash Table

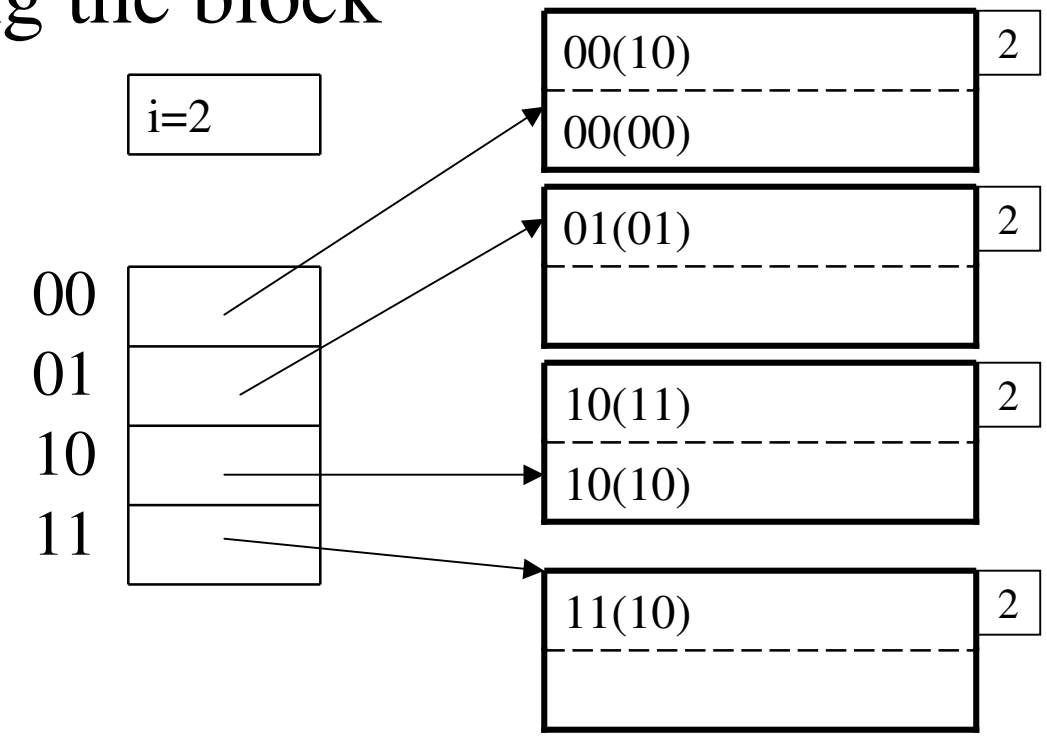
- Now insert 0000, then 0101



- Need to split block

Insertion in Extensible Hash Table

- After splitting the block



Performance Extensible Hash Table

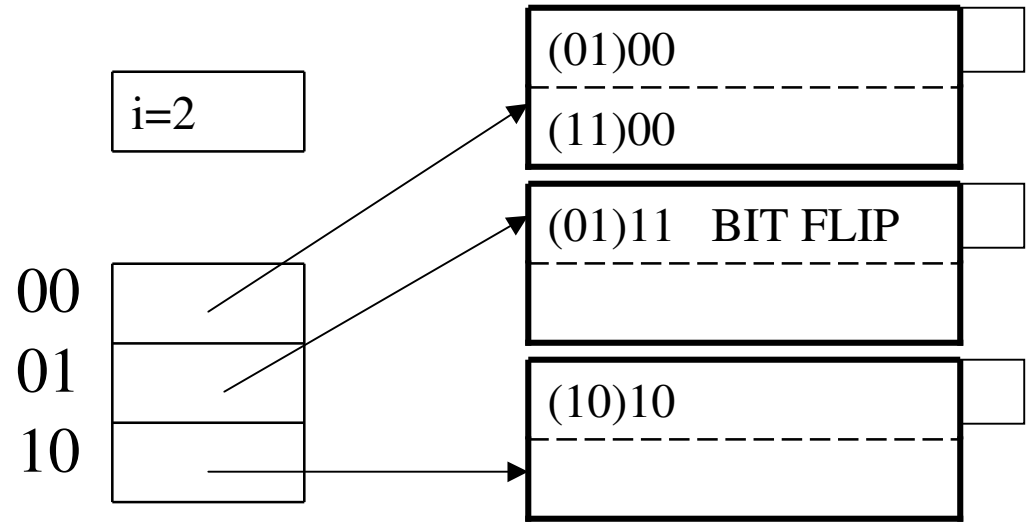
- No overflow blocks: access always one read
  - However, collisions (when hash values are identical) may require overflow blocks
- BUT:
  - Extensions can be costly and disruptive
  - After an extension table may no longer fit in memory
  - Performs poorly on skewed data (but good hash functions usually take care of that)

Linear Hash Table

- Idea: extend only one entry at a time
- Problem: n= no longer a power of 2
- Let i be such that $2^i <= n < 2^{i+1}$
- After computing h(k), use last i bits:
  - If last i bits represent a number > n, change msb from 1 to 0 (get a number <= n)

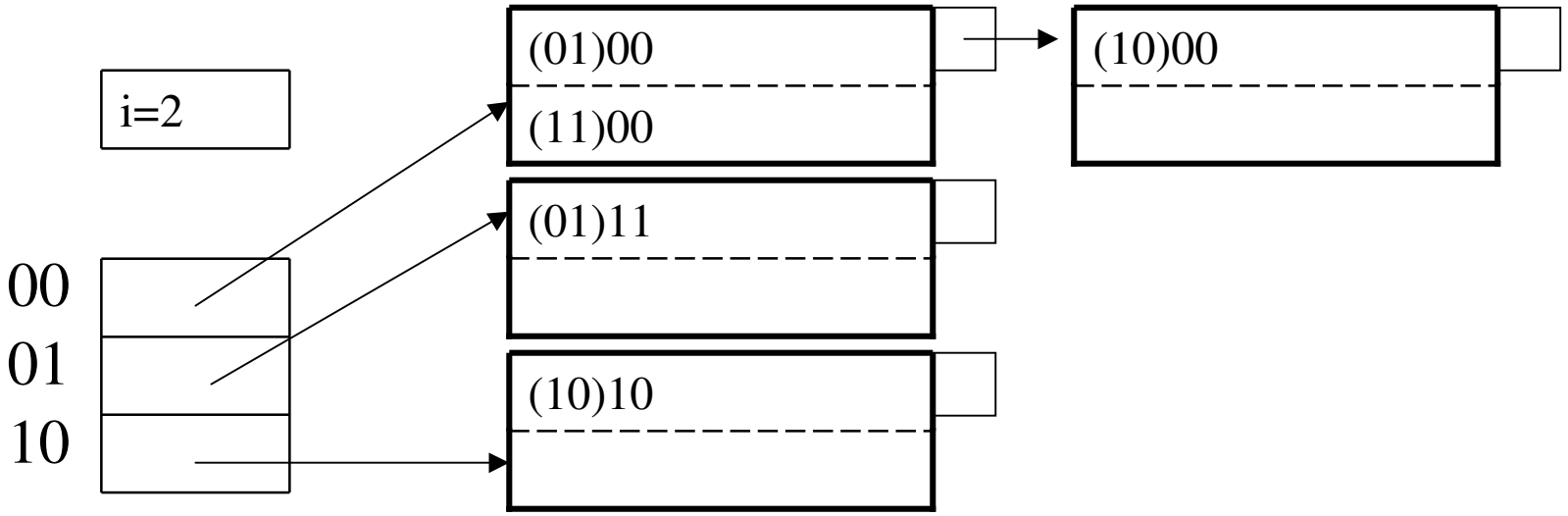
Linear Hash Table Example

- n=3



Linear Hash Table Example

- Insert 1000: overflow blocks…

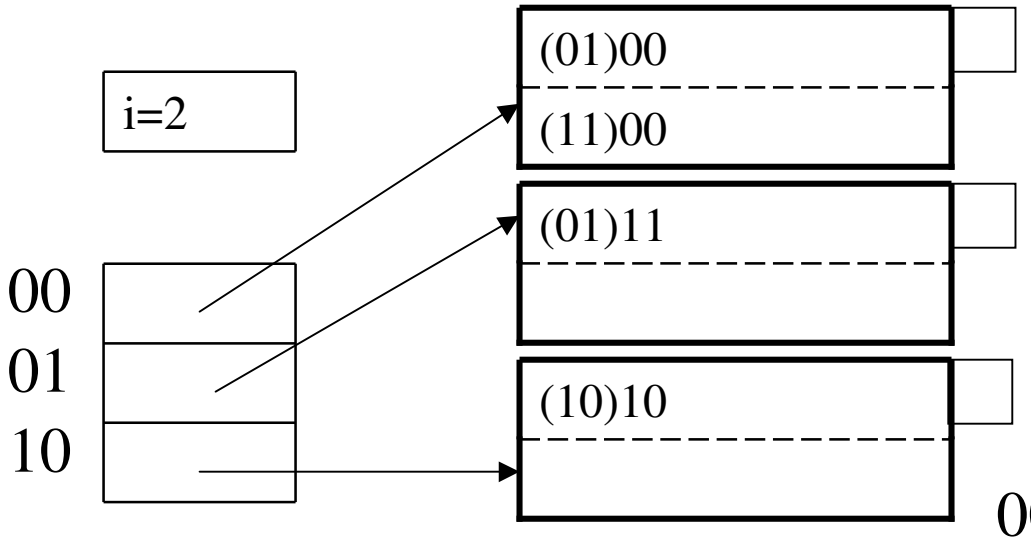


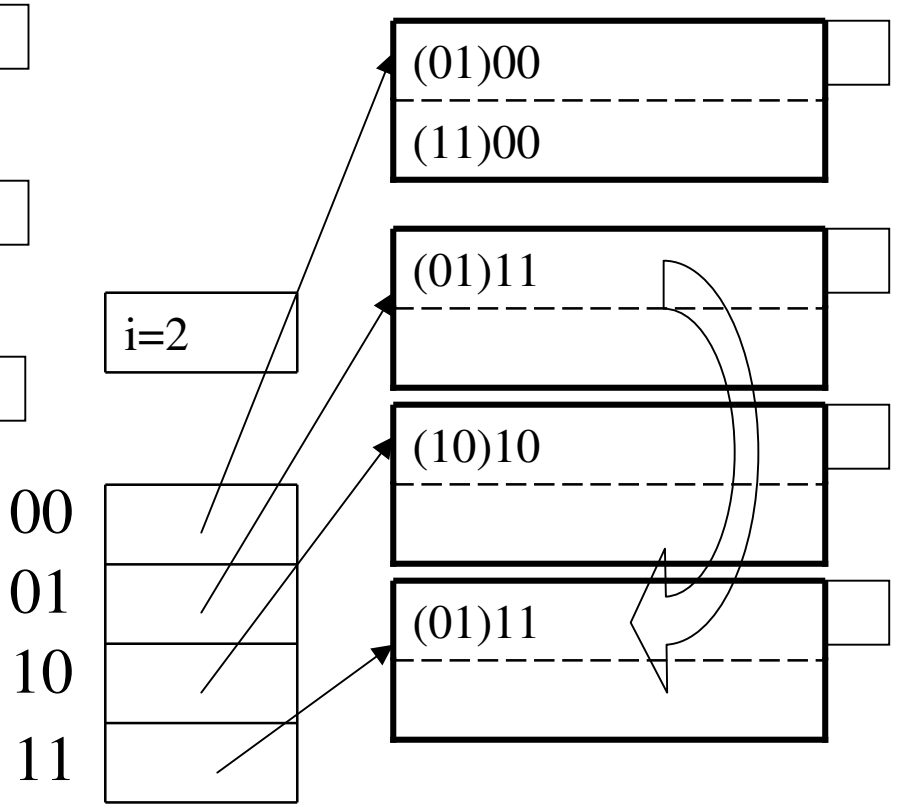
Linear Hash Tables

- Extension: independent on overflow blocks
- Extend n:=n+1 when average number of records per block exceeds (say) 80%

Linear Hash Table Extension

- From n=3 to n=4



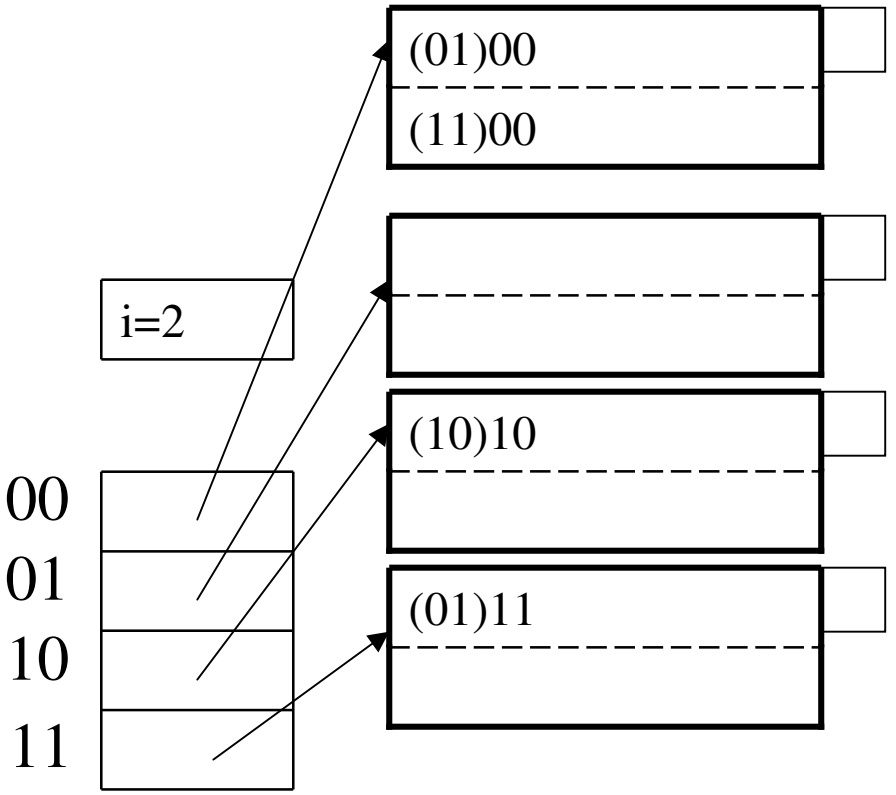


Only need to touch
one block (which one ?)

Linear Hash Table Extension

- From n=3 to n=4 finished

- Extension from n=4 to n=5 (new bit)
- Need to touch every single block (why ?)



Optimization Overview

- S. Chaudhuri, *An overview of query optimization in relational systems*
- Three components:
  - A search space (given by *algebraic laws*)
  - A cost estimation technique
  - An enumeration algorithms
- Two philosophies:
  - Heuristics-based optimizations
  - Cost-based optimizations

Algebraic Laws

- Commutative and Associative Laws
  - $R \cup S = S \cup R,\ R \cup (S \cup T) = (R \cup S) \cup T$
  - $R \cap S = S \cap R,\ R \cap (S \cap T) = (R \cap S) \cap T$
  - $R \bowtie S = S \bowtie R,\ R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
- Distributive Laws
  - $R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T)$

Algebraic Laws

- Laws involving selection:
  - $\sigma_{C \text{ AND } C'}(R) = \sigma_{C}(\sigma_{C'}(R)) = \sigma_{C}(R) \cap \sigma_{C'}(R)$
  - $\sigma_{C \text{ OR } C'}(R) = \sigma_{C}(R) \cup \sigma_{C'}(R)$
  - $\sigma_{C}(R \bowtie S) = \sigma_{C}(R) \bowtie S$
    - When C involves only attributes of R
  - $\sigma_{C}(R - S) = \sigma_{C}(R) - S$
  - $\sigma_{C}(R \cup S) = \sigma_{C}(R) \cup \sigma_{C}(S)$
  - $\sigma_{C}(R \cap S) = \sigma_{C}(R) \cap S$

Algebraic Laws

- Example: R(A, B, C, D), S(E, F, G)
  - $\sigma_{F=3}(R \underset{D=E}{\bowtie} S) =$?
  - $\sigma_{A=5 \text{ AND } G=9}(R \underset{D=E}{\bowtie} S) =$?

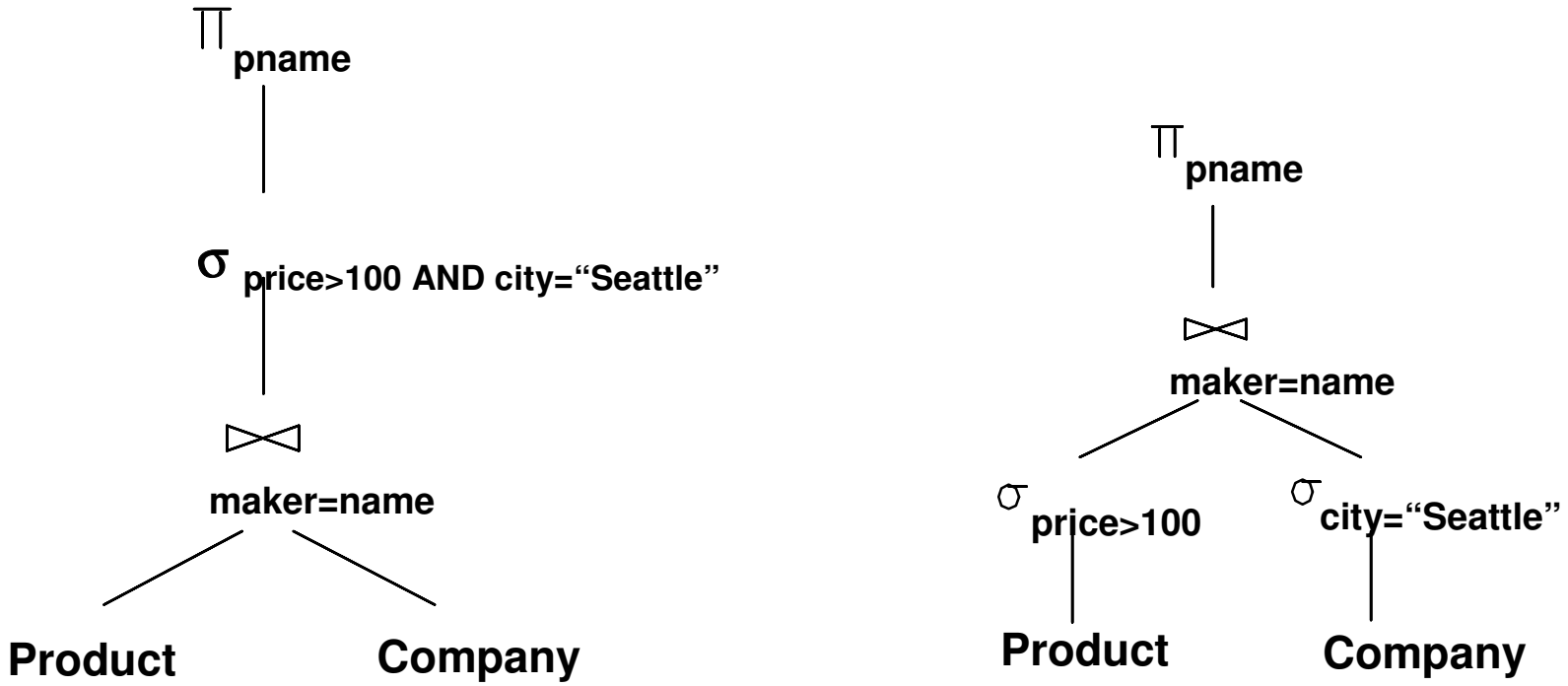
Algebraic Laws

- Laws involving projections
  - $\Pi_M(R \bowtie S) = \Pi_N(\Pi_P(R) \bowtie \Pi_Q(S))$
    - Where N, P, Q are appropriate subsets of attributes of M
  - $\Pi_M(\Pi_N(R)) = \Pi_{M,N}(R)$
- Example R(A,B,C,D), S(E, F, G)
  - $\Pi_{A,B,G}(R \underset{D=E}{\bowtie} S) = \Pi_{?}(\Pi_{?}(R) \underset{D=E}{\bowtie} \Pi_{?}(S))$

Heuristic Based Optimizations

- Query rewriting based on algebraic laws
- Result in better queries most of the time
- Main heuristics:
  - Push selections down the tree

Heuristic Based Optimizations



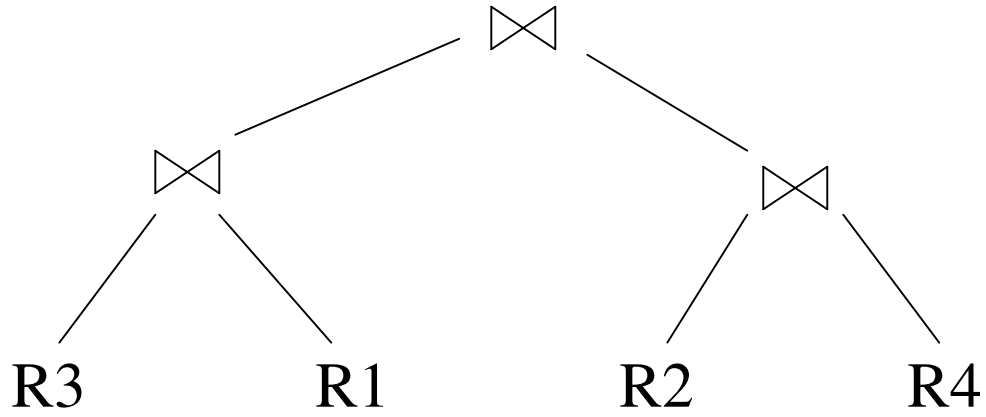
The earlier we process selections, less tuples we need to manipulate higher up in the tree (but may cause us to indexes).

Cost-Based Optimization

- Main optimization unit:
  - set of joins, i.e. single select-from-where block
  - Hence: *the join reordering problem*
- Optimization methods:
  - Dynamic programming (System R, 1977), for joins:
    - Conceptually cleanest
  - Rule-based optimizations, for arbitrary queries:
    - Volcano → SQL server
    - Starburst → DB2

Join Trees

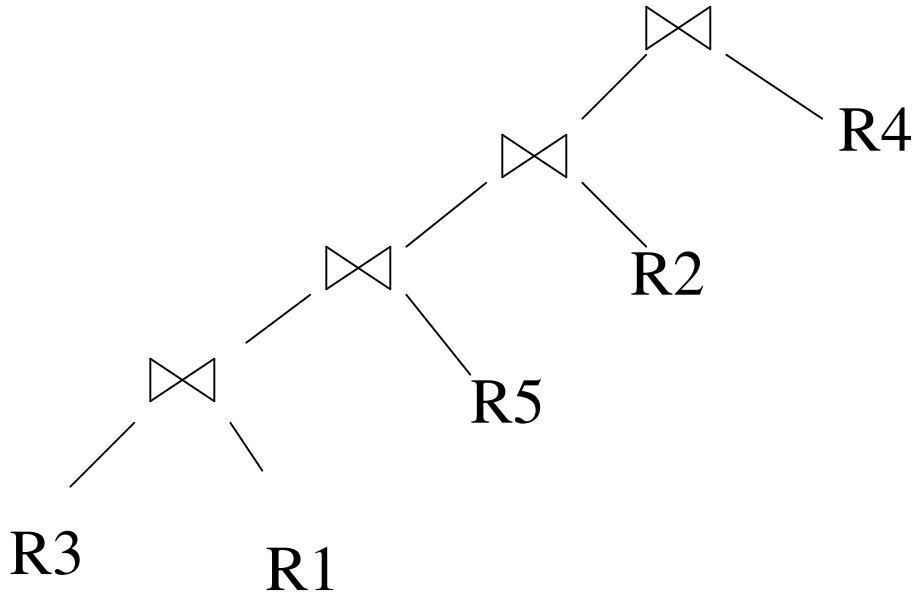
- R1 ⋈ R2 ⋈ …. ⋈ Rn
- Join tree:



- A join tree represents a plan. An optimizer needs to inspect many (all ?) join trees

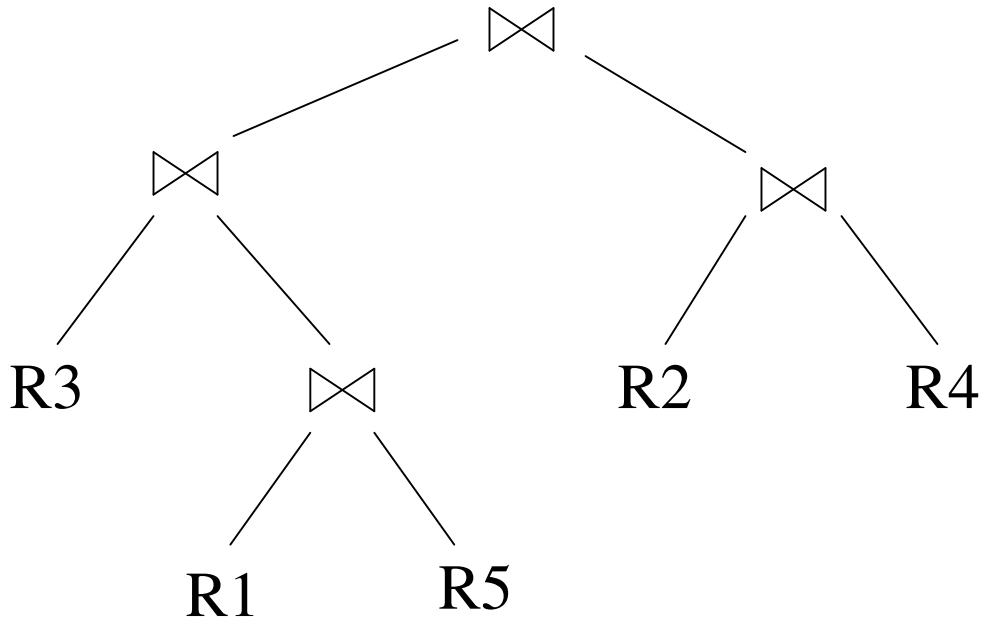
Types of Join Trees

- Left deep (or left-linear):



Types of Join Trees

- Bushy:



Problem

- Given: a query $R1 \bowtie R2 \bowtie \ldots \bowtie Rn$
- Assume we have a function cost() that gives us the cost of every join tree
- Find the best join tree for the query

Justification for Handling Joins Separately

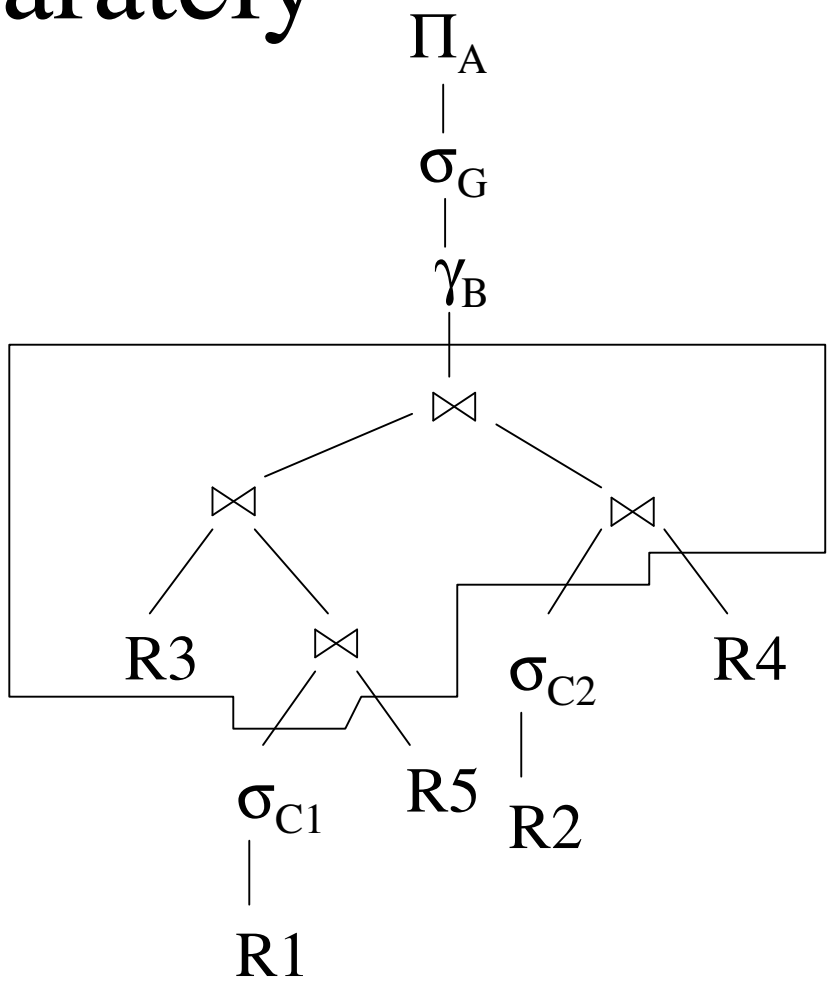
Select A

From R_1, R_2, …, R_n

Where C

GroupBy B

Having G



Dynamic Programming

- Idea: for each subset of {R1, …, Rn}, compute the best plan for that subset
- Compute best plan as follows:
  - Step 1: for {R1}, {R2}, …, {Rn}
  - Step 2: for {R1,R2}, {R1,R3}, …, {Rn-1, Rn}
  - …
  - Step n: for {R1, …, Rn}
- A subset of {R1, …, Rn} is also called a *subquery*

Dynamic Programming

- For each subquery Q $\subseteq$ {R1, …, Rn} compute the following:
  - Size(Q)
  - A best plan for Q: Plan(Q)
  - The cost of that plan: Cost(Q)
- Additional complication:
  - Consider *interesting orders*
  - For each subquery Q $\subseteq$ {R1, …, Rn}, generate one plan for each interesting order

Dynamic Programming

- Step 1: For each {Ri} do:
  - Size({Ri}) = B(Ri)
  - Plan({Ri}) = Ri
  - Cost({Ri}) = (cost of scanning Ri)

Dynamic Programming

- Step i: For each Q $\subseteq$ {R1, …, Rn} of cardinality i do:
  - Compute Size(Q) (later…)
  - For every pair of subqueries Q', Q''
    s.t. Q = Q' U Q''
    compute cost(Plan(Q') $\bowtie$ Plan(Q''))
  - Cost(Q) = the smallest such cost
  - Plan(Q) = the corresponding plan

Dynamic Programming

- Return Plan({R1, …, Rn})

Dynamic Programming

Heuristics for Reducing the Search Space

- Restrict to left linear trees
- Restrict to trees “without cartesian product”

Rule-based Optimizations

- Volcano:
  - Main idea: let programmers define rewrite rules, based on the algebraic laws
  - System searches for “best plan” by applying laws repeatedly
  - Need to avoid cycles, etc.
  - Join-reordering becomes harder, but can handle other operators too
- Starburst:
  - Same, but keep larger nodes, corresponding to one select-from-where block
  - Apply rewrite rules inter-blocks
  - Do dynamic programming inside blocks