Query Processing

These slides are mostly taken verbatim, or with minor changes, from those prepared by Stephen Hegner (http://www.cs.umu.se/ hegner/) of Umeå University

Overview

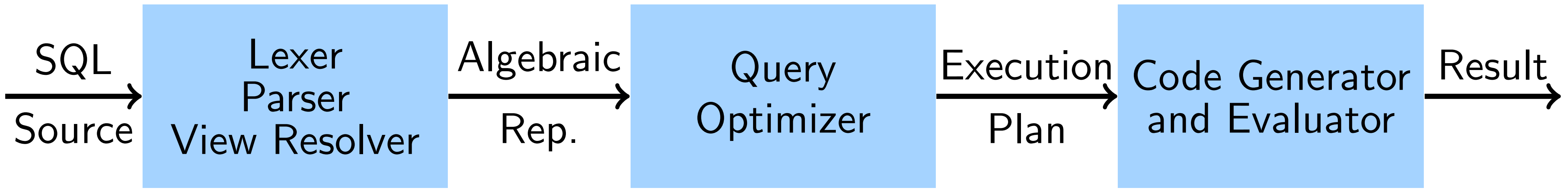
Question: How is a query in SQL processed by the system to produce the answer?

- The block diagram below identifies the three main steps.

Lexer + Parser + View Resolver: In the first step, the (declarative) SQL query is translated to an operational query, usually but not always expressed in the extended relational algebra.

Query Optimizer: In the second step, the algebraic representation is augmented to be an *execution plan*, with possible rearrangement of the algebraic operations as well as implementation-specific information on how to evaluate in an efficient fashion.

Code Generator and Evaluator: In the final step, the plan of execution is carried out.



Scanning, Parsing, and View Resolution

- Lexical analysis and parsing are carried out using well-known techniques from translator design.

Lexical analysis: The *lexer* (or *lexical analyzer* or *tokenzier*) breaks the input stream up into *tokens*.

Parsing: The *parser* builds a *parse tree* for the tokens according to a grammar for the language.

- These topics will not be considered in this course.

View resolution: corresponds to the generation of an *intermediate representation* in programming-language translation.

- In the case of SQL, the extended relational algebra is often but not always used the the intermediate language.
- The relationship between SQL and the extended relational algebra, as well as how to translate queries from one to the other, has already been studied in the introductory course.
- The topic of how to obtain an equivalent expression in the extended relational algebra from a query in SQL will not be considered further here.

Query Optimization

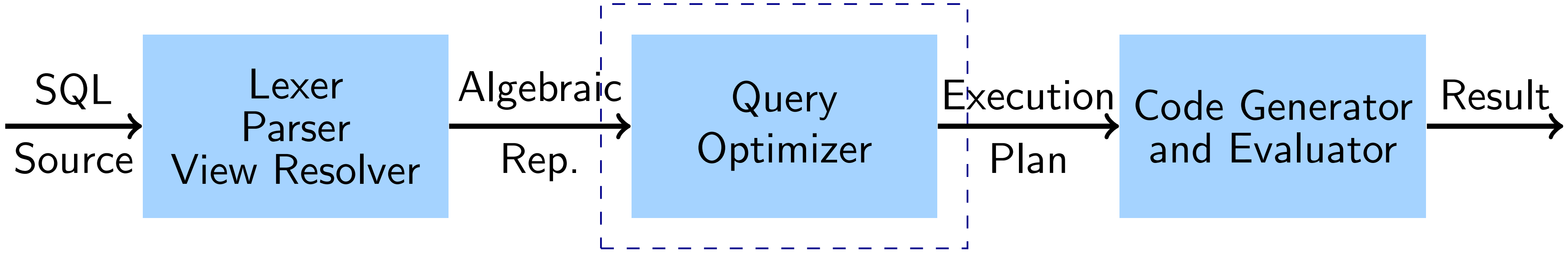
- Query optimization involves at least two distinct processes.

Query rewriting: A given expression in the relational algebra (*e.g.*, the output of the lexer + parser + view resolver) may be represented by an equivalent expression which is amenable to more efficient evaluation.

Annotation: An expression in the extended relational algebra may be annotated with specific information on how to carry out its steps, such as:

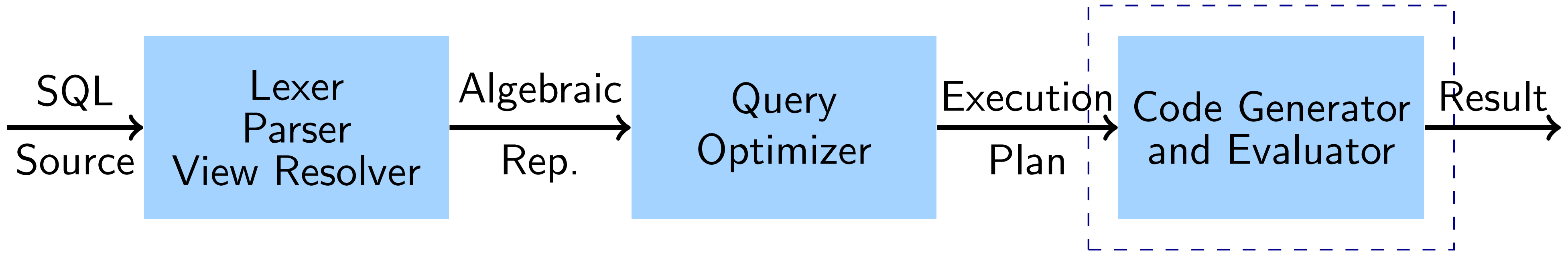
- which algorithms to use;
- which indices to use.

- These topics will be examined in these slides.



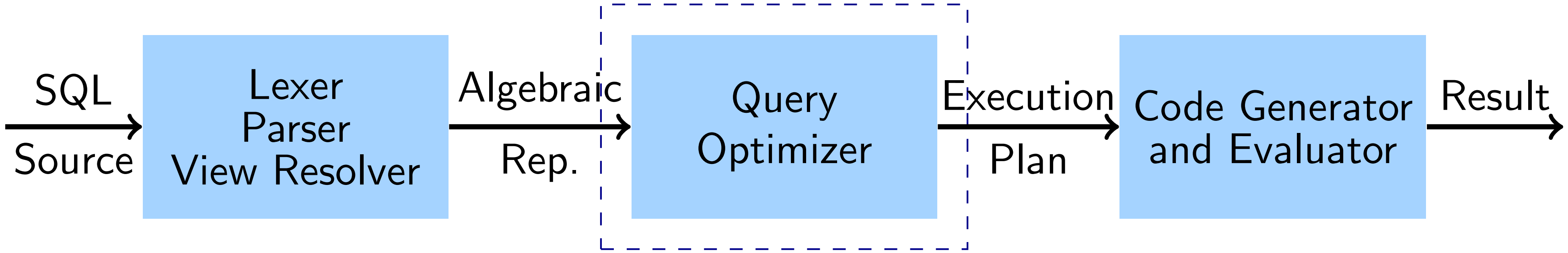
Code Generation and Evaluation

- Executing an execution plan is a relatively straightforward process, although there are certainly nontrivial details which must be addressed.
- This task will not be examined further in these slides.
- Thus, the focus will be upon query optimization.



Code Generation and Evaluation

- Executing an execution plan is a relatively straightforward process, although there are certainly nontrivial details which must be addressed.
- This task will not be examined further in these slides.
- Thus, the focus will be upon query optimization.



Executing Operations in the Extended Relational Algebra

- In these slides, the extended relational algebra will be used as the intermediate language for (unannotated) execution plans.
- Therefore, it is important to begin with a study of algorithms for the following, under a variety of conditions for index availability.

  projection

  selection

  join

  removal of duplicates

  ordering of results

  aggregation

Basic Measures of Cost for Data Access

- Recall that access to secondary storage (usually hard disks) takes much longer (thousands of times longer) than access to primary storage (main memory).
- Minimizing the number of times that secondary storage must be accessed is therefore paramount in the design of efficient algorithms for query processing.
- It is useful to begin with some basic parameters for disk access.

t_S: The (average) time required to access one block of data (seek time + rotational latency).

t_T: The (average) time required to transfer one block of data from secondary to primary storage.

- An operation which requires n_s seeks to transfer n_b blocks thus requires a total time of $n_s \cdot t_S + n_b \cdot t_T$.
- The relationship between n_s and n_b depends upon how the required blocks are arranged on the secondary device (random vs. sequential neighbors).

Cost Measures Associated with Indices

- If an attribute is indexed and a query involves that attribute, then it is often the case that an optimal evaluation algorithm will involve access via that attribute.

h_i: For an index which is a B$^+$-tree, the depth of the <u>index</u>; *i.e.*, one less than the length of a path from the root to a leaf.

- As noted previously, it is access to secondary storage which is the prime consumer of time.
- Therefore, it is appropriate to decompose

$$h_i = h_{i_{\text{pri}}} + h_{i_{\text{sec}}}$$

as follows.

$h_{i_{\text{pri}}}$: In a path from the root to a leaf, the (average) number of pointers whose destination is already in main memory.

$h_{i_{\text{sec}}}$: In a path from the root to a leaf, the (average) number of pointers whose destination is not in main memory.

Disk access rule of thumb: In most cases, access times along $h_{i_{\text{pri}}}$ may be ignored, since they will be thousands of times less than for $h_{i_{\text{sec}}}$.

Cases for Selection

- Selection is the most basic operation of the extended relational algebra which involves the use of indices.
- It is convenient to decompose access into a number of cases.

Simple cases: A is an attribute and e is a fixed expression which may be evaluated in constant time.

Equality on a single attribute: $\sigma_{A=e}$

Inequality on a single attribute: $\sigma_{A \neq e}$

Simple range on a single attribute: $\sigma_{A \leq e}$, $\sigma_{A \geq e}$, $\sigma_{A < e}$, $\sigma_{A > e}$

Compound cases: θ and each θ_i is a simple condition $A_i \odot e_i$ with $\odot \in \{=, \leq, <, \geq, >, \neq\}$.

Conjunction: $\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_k}$

Disjunction: $\sigma_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_k}$

Negation: $\sigma_{\neg\theta}$ (Not really needed as a separate case, since, for example $\sigma_{\neg(A \leq e)}$ is the same as $\sigma_{A > e}$, but care must be taken to handle null values correctly.)

Simple Selection without Index Support

n_{blk}: The number of blocks used for the file containing the relation.

A1: Linear search: The entire relation is searched for tuples which match the condition $A \odot e$, where $\odot \in \{=, \leq, <, \geq, >, \neq\}$.

Total cost (time): $n_{\text{S}} \cdot t_S + n_{\text{blk}} \cdot t_T$.

- n_{S} is the number of distinct seeks required.
- In the best case the data are stored contiguously on disk and $n_{\text{S}} = 1$, but this cannot be guaranteed in general.
- $n_{\text{S}} \leq n_{\text{blk}}$ always holds.

A1: Linear search; equality on a key: The entire relation is searched for tuples which match the condition $A = e$.

Worst-case total cost (time): $n_{\text{S}} \cdot t_S + n_{\text{blk}} \cdot t_T$.

Best-case total cost (time): $t_S + t_T$.

Average-case total cost (time): $(n_{\text{S}} \cdot t_S + n_{\text{blk}} \cdot t_T)/2$.

- On the average, about half of the file must be searched before the key is found.

Simple Selection with Primary Index Support

A2: Primary B$^+$-tree index on a key; equality search on the index attribute:

This is the best of all possible cases.

All-cases total cost (time): $(h_{i_{\text{sec}}} + 1) \cdot (t_S + t_T)$.

- There is one seek plus disk access for each level of the index which is not in main memory, plus one more to reach the block containing the desired record

A3: Primary B$^+$-tree index on a non-key; equality search on index attribute:

This is almost as good, but more than one block may need to be retrieved.

All-cases total cost (time): $(h_{i_{\text{sec}}} + 1) \cdot t_S + (h_{i_{\text{sec}}} + n_{\text{rec}}) \cdot t_T$.

- The number of seeks is exactly as in the previous case.
- The number of transfers is determined by the number n_{rec} of blocks which contain records matching the key.

- The times will be even less if the block and/or more of the index is cached.

Simple Selection with Secondary Index Support

A4: Secondary B$^+$-tree index on key; equality search on index attribute:

The analysis is similar to the case for a primary index, since only one record is retrieved.

All-cases total cost (time): $(h_{i_{\text{sec}}} + 2) \cdot (t_S + t_T)$.

- There is one more seek+access than for a primary index — the leaves of the B$^+$-tree will contain references, not actual records.

A4: Secondary B$^+$-tree index on non-key; equality search on index attribute:

Here the records to be found need not lie in the same block, or even in contiguous blocks.

Worst-case total cost (time): $(h_{i_{\text{sec}}} + 1 + n_{\text{rec}}) \cdot (t_S + t_T)$.

- n_{rec} is the number of records which are retrieved.
- There is one seek and one access for each such record.
- There is one additional seek+access per record because the leaves of the B$^+$-tree will contain references and not the actual records.
- The average case is likely not much better.

- The times will be less if the blocks and/or more of the index is cached.

Extension to Inequality and Simple Range Queries

A5: Primary B$^+$-tree index; simple range search on the index attribute:

Applies to both key and non-key attributes.

All-cases total cost (time): $(h_{i_{\mathrm{sec}}} + 1) \cdot t_S + (h_{i_{\mathrm{sec}}} + n_{\mathrm{rec}}) \cdot t_T$.

- This time is identical to that of A3, but n_{rec} will of course be larger in general.
- The retrieved records will be stored contiguously.

Example: $\sigma_{A \geq 10}$. Here the retrieval starts at $A = 10$, and returns all blocks "to the right" in the sequential ordering.

A6: Secondary B$^+$-tree index; simple range search on the index attribute:

Applies to both key and non-key attributes.

Worst-case total cost (time): $(h_{i_{\mathrm{sec}}} + 1 + n_{\mathrm{rec}}) \cdot (t_S + t_T)$.

- This time is identical to that of A4, but n_{rec} will of course be larger in general.
- The retrieved records will not be stored contiguously.
- The average case is likely not much better.

Extension to Conjunctive Selection Conditions

- Considered here are queries of the form $\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_k}$

A7: Conjunctive selection using one B$^+$-tree index:

- This approach requires that (at least) one of the conditions θ_i involve an indexed attribute.
- First, evaluate σ_{θ_i} using one of the approaches A2-A6.
- Then, resolve the remaining conditions directly on the result of the above evaluation.
- If there are alternatives for θ_i, two heuristics may apply.
  - Choose the one which will return the fewest records for σ_{θ_i}.
  - Choose the one which is fastest for answering σ_{θ_i}.

A8: Conjunctive selection using a composite B$^+$-tree index:

- Here there is a *composite index* on two of the attributes, one for θ_i and a second for θ_j.
- The records which satisfy both conditions may be retrieved at once.
- The details are similar to those of A7 and will not be developed in detail here.

Extension to Conjunctive Selection Conditions – 2

- Considered here are queries of the form $\sigma_{\theta_1 \wedge \theta_2 \wedge \ldots \wedge \theta_k}$

A9: Conjunctive selection by intersection of pointers or identifiers:

- Let $J \subseteq \{1, 2, \ldots, k\}$ with the property that for each $j \in J$, it is possible to retrieve a set of pointers or identifiers which provides the solution to σ_{θ_j}.
- Then the intersection of these sets provides pointers or identifiers to a solution of $\sigma_{\bigwedge_{j \in J} \theta_j}$.
- The resolution of the remaining subqueries of the form σ_{θ_i} with $i \notin J$ is achieved by examining the retrieved records directly, as in the case of A7.
- The computation of intersection will be considered shortly.

Extension to Disjunctive Selection Conditions

- Considered here are queries of the form $\sigma_{\theta_1 \vee \theta_2 \vee \ldots \vee \theta_k}$
- As is the case with most other problems involving disjunction (*e.g.*, satisfiability of logical expressions), there are in general no efficient algorithms.
- One approach which may provide some improvement is the disjunctive version of A9.

A10: Disjunctive selection by union of pointers or identifiers:

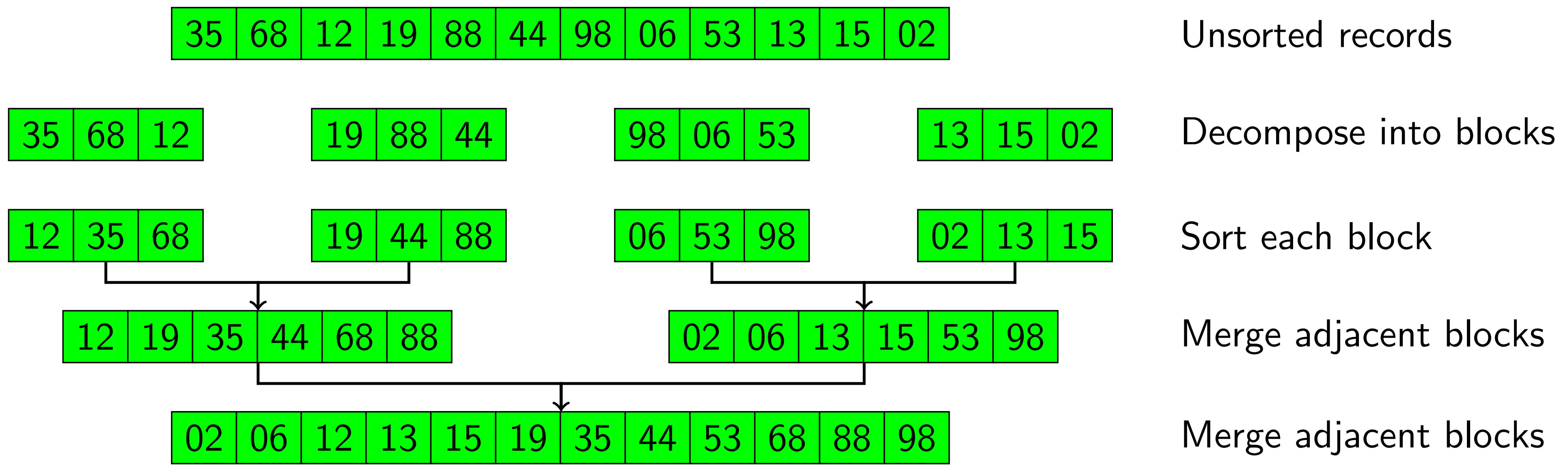
- The idea here is to retrieve a set of pointers or identifiers, one for each query σ_{θ_i},
- This approach only works is such a retrieval is possible for all indices.
- In the notation of A9, J must equal $\{1, 2, \ldots, k\}$.
- Then the union of these sets provides pointers or identifiers to the desired result.
- The computation of union will be considered shortly.

Two-Way Sort-Merge

- It is often the case that the set of records to be sorted is too large to fit into main memory, so a special algorithm is needed.

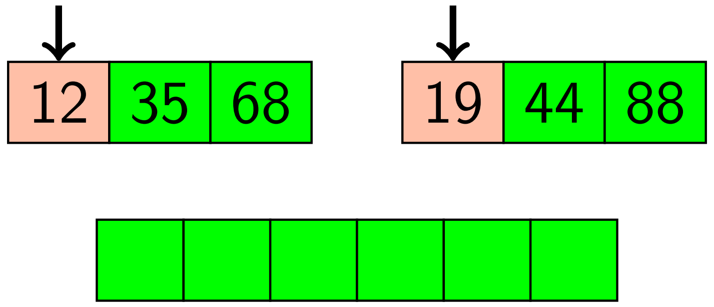
Sort-merge: The idea behind the sort-merge algorithm is shown below.

- First the unsorted list of records is decomposed into blocks which are small enough to fit in main memory (one at a time).
- The blocks are next brought into memory and sorted, one at a time.
- The blocks are finally merged repeatedly until a single list is obtained.



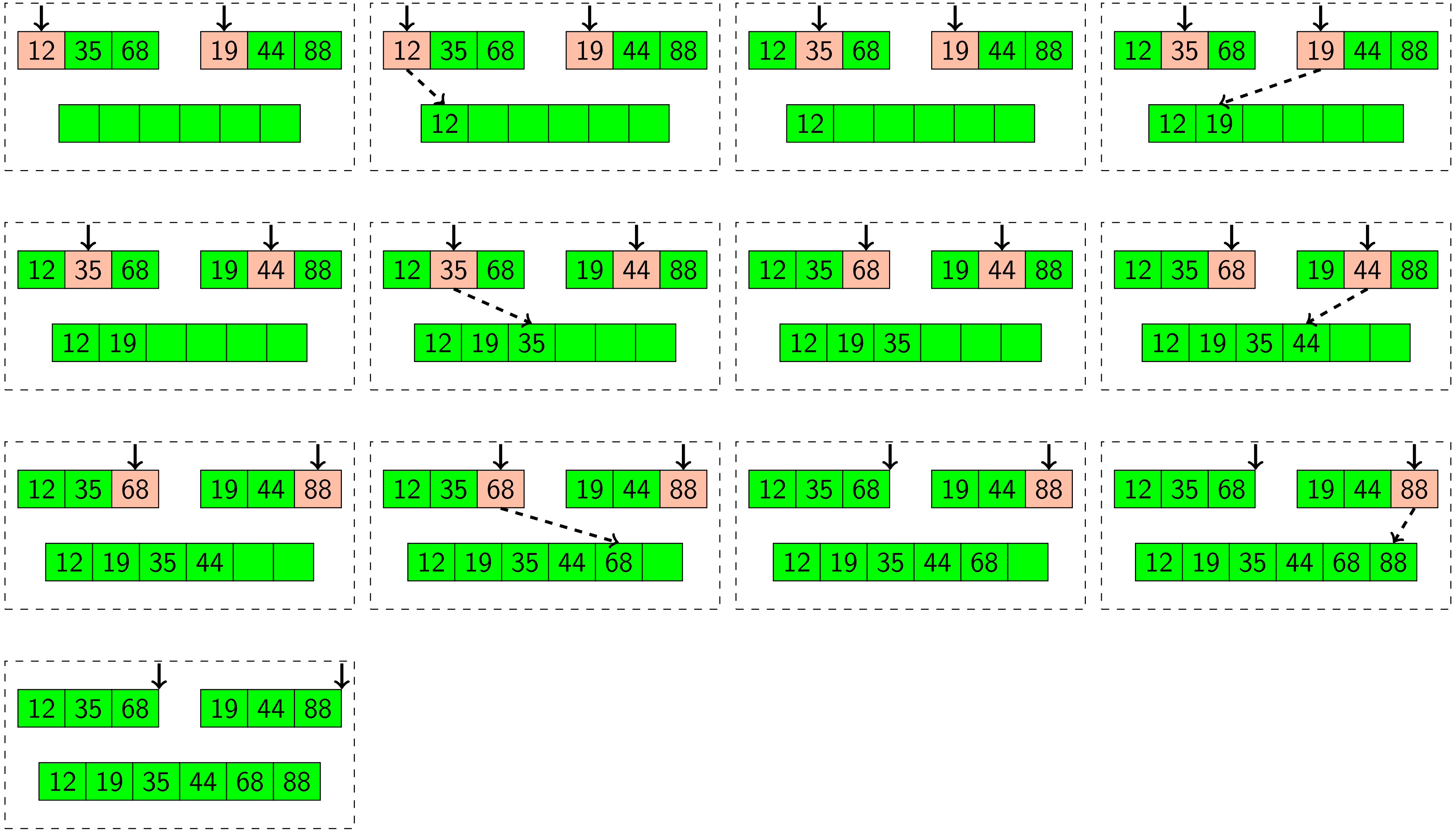
Two-Way Sort-Merge with Limited Primary Memory

- The algorithm just described involves the recursive merger of two potentially large lists, too large to fit into primary memory.
- The algorithm for execution in limited memory is as follows.
- Only those records shown in coral need be in main memory.
  - In general, for efficiency, much larger blocks, as opposed to just one record, will be brought into memory at once.
- A pointer is kept to the first unused element in each list, and is advanced as the elements are merged into the new list.
- The complete processing for this tiny example is shown on the next slide.



Two-Way Sort-Merge with Limited Primary Memory – Example

- Steps are left to right, then top to bottom.



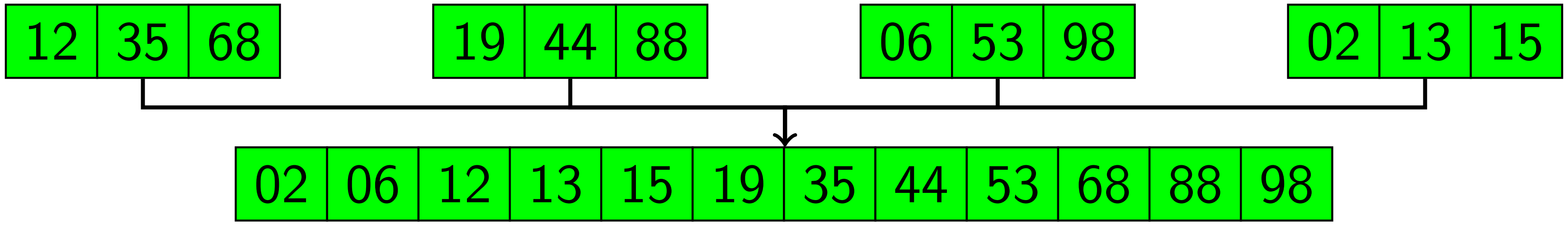
N-Way Sort-Merge

Two-way merging: The examples just shown merge two lists a time; hence the name.

N-way merging: It is certainly possible to merge more than two lists a time.

- When *N* lists are merged at a time, it is called *N*-way merging.

- A simple example of four-way merging is shown below.
- Regardless of the number of ways, the number of blocks which may be kept in main memory at one time is fixed.
- With *N*-way merging, the number of blocks need not be a power of two.



Complexity of the Algorithm for External Sorting

n_{blk}: Total number of blocks of records.

n_{bps}: Number of blocks transferred between primary and secondary memory in one operation (requires only one seek).

M: Let M denote the number of blocks which will fit into the assigned buffer area of main memory ($n_{\text{bps}} < M$).

Order of Complexity (number of seeks): After stripping away constants and smaller terms, the order of the number of seeks looks like this:

$$\left\lceil \frac{n_{\text{blk}}}{n_{\text{bps}}} \right\rceil \cdot \left(\left\lceil \log_{\lfloor M/n_{\text{bps}} \rfloor} \left(\frac{n_{\text{blk}}}{M} \right) \right\rceil \right)$$

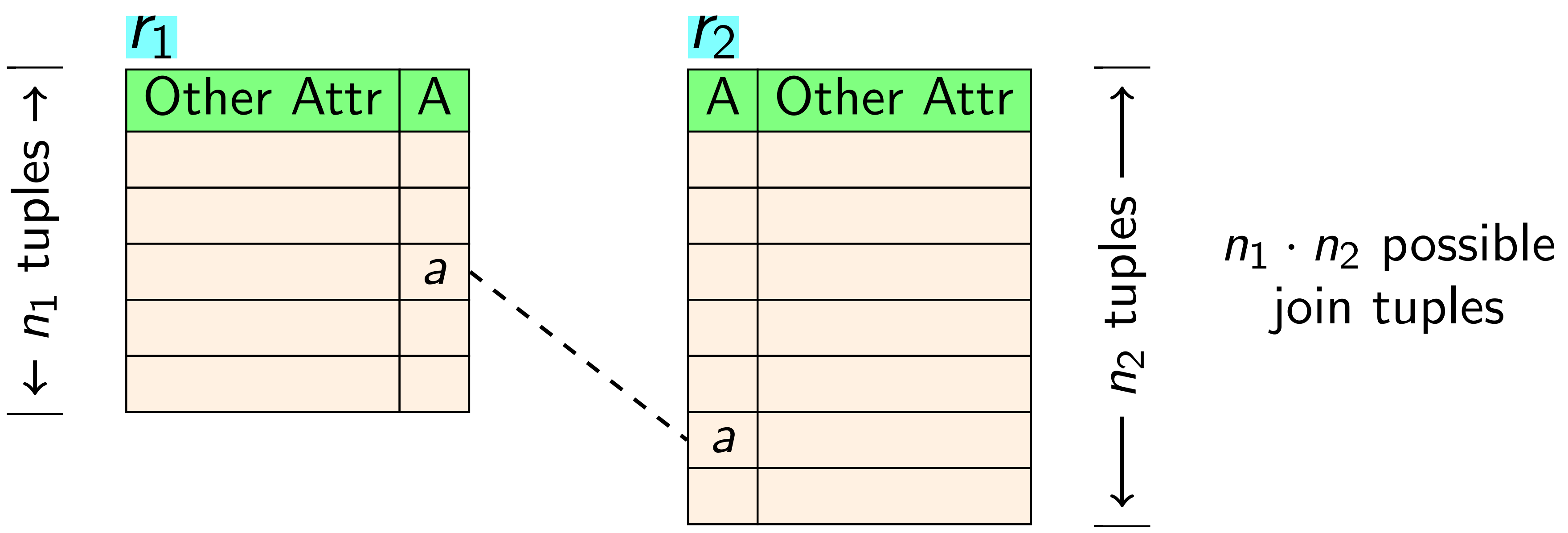
Observations: The smaller the ratios $n_{\text{blk}}/n_{\text{bps}}$, n_{blk}/M, and M/n_{bps}, the better.

The Nested-Loop Algorithm for Join

- Algorithms for computing the join are particularly important since the number of records to process is the product of the number records in each of the two relations.

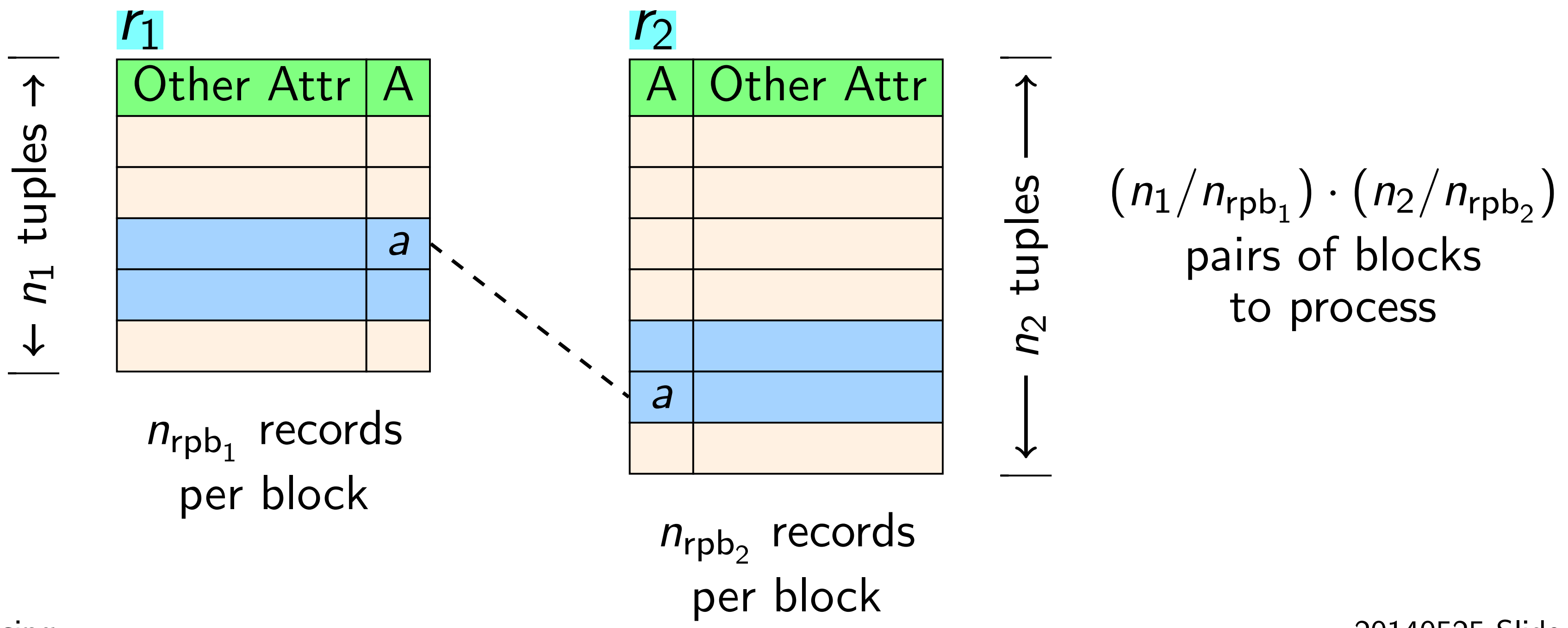
Nested loop: The brute-force nested loop approach simply compares each tuple in the first relation r_1 to each tuple in the second r_2.

- In the worst case, this can result in $n_1 \cdot n_2$ disk seeks, which is prohibitive for all but the smallest relations.


r_1
Other Attr
A
a
$\leftarrow n_1$ tuples $\rightarrow$
r_2
A
Other Attr
a
$\leftarrow n_2$ tuples $\rightarrow$
$n_1 \cdot n_2$ possible join tuples

The Block-Oriented Nested-Loop Algorithm for Join

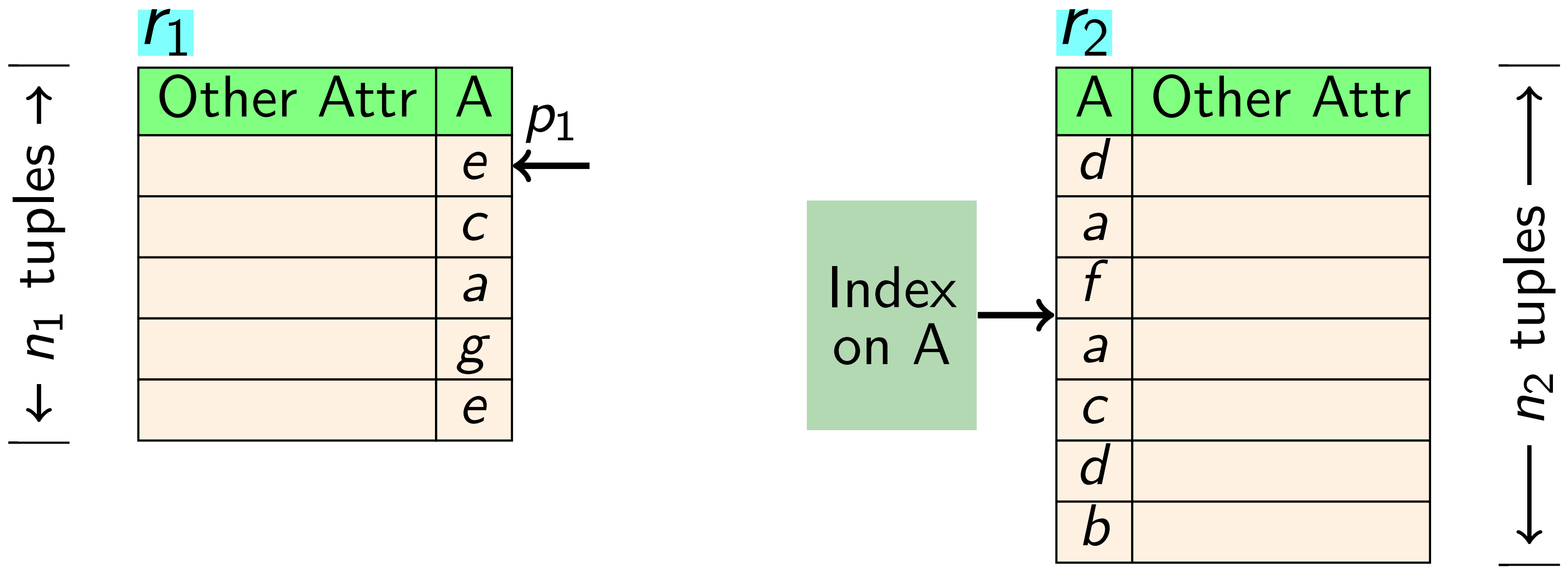
- The basic nested-loop algorithm can be improved by retrieving large blocks of records from each relation at once.
- While the number of comparisons is not changed, the number of disk seeks is reduced greatly.
- This will result in $(n_1/n_{\mathrm{rpb}_1}) \cdot (n_2/n_{\mathrm{rpb}_2})$ disk seeks, which may or may not be prohibitive, depending upon the number of blocks.
- These algorithms are very basic, and for many cases better ones are available.



r_1
Other Attr
A
a
$\leftarrow n_1$ tuples $\rightarrow$
n_{rpb_1} records per block
r_2
A
Other Attr
a
$\leftarrow n_2$ tuples $\rightarrow$
n_{rpb_2} records per block
$(n_1/n_{\mathrm{rpb}_1}) \cdot (n_2/n_{\mathrm{rpb}_2})$ pairs of blocks to process

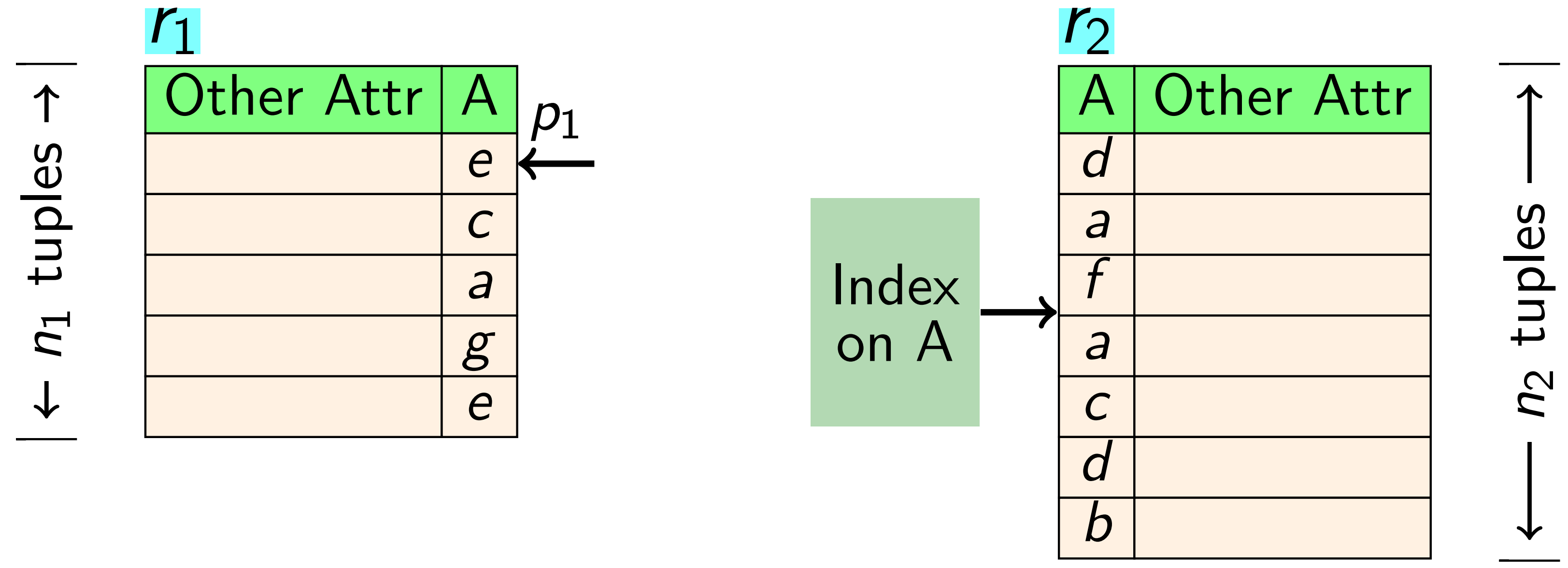
Join with an Index on One of the Relations

- Suppose that an index exists on the join attribute(s) of one of the relations (r_2 here).
- The other relation (r_1) has no index and the tuples need not be ordered on the join attribute(s).
- The tuples of r_1 may be processed sequentially, looking in the index to see if a matching tuple exists in r_2.


r_1
$\leftarrow n_1$ tuples $\rightarrow$
Other Attr
A
e
c
a
g
e
p_1
r_2
Index on A
A
Other Attr
d
a
f
a
c
d
b
$\leftarrow n_2$ tuples $\rightarrow$

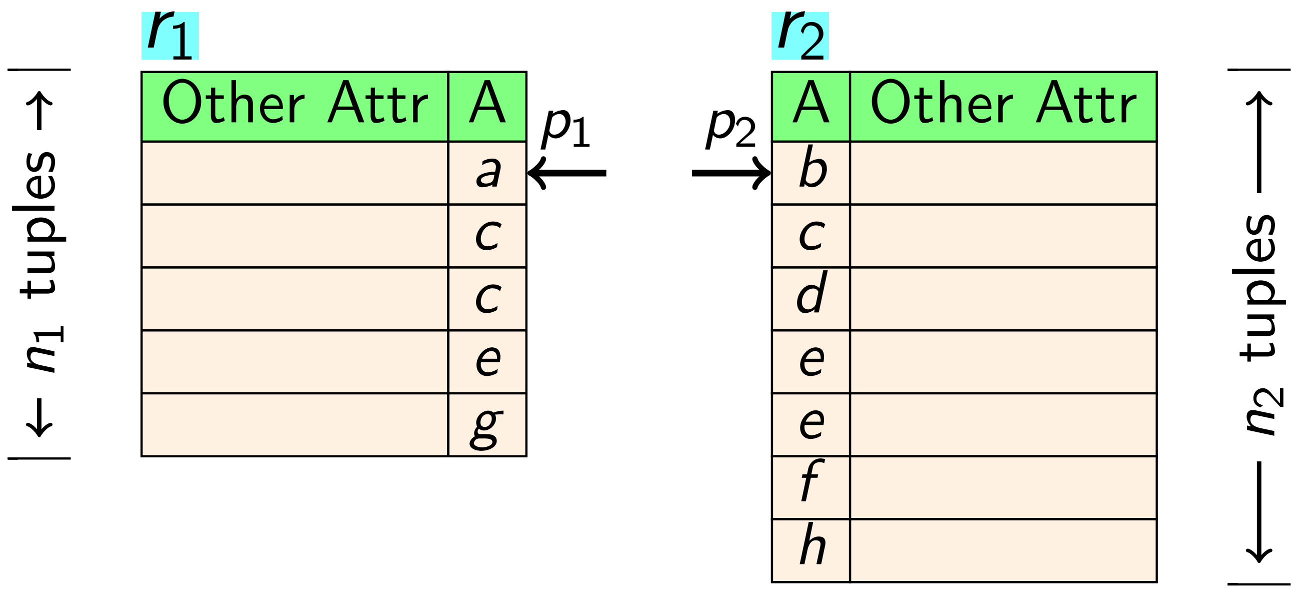
Join with an Index on One of the Relations — Complexity

- The complexity in this case is formally $\Theta(n_1)$, assuming that an index lookup in r_2 takes constant time.
- If index lookup in r_2 takes $\Theta(\log(n_2))$ time, then the complexity becomes $\Theta(n_1 \cdot \log(n_2))$.
- However, the constants are likely to be quite large, since disk seeks will be involved.
- The performance may of course be improved by retrieving large blocks of tuples from r_1 at a a time, and batching identical values for A with a single index search in r_2.



Merge Join

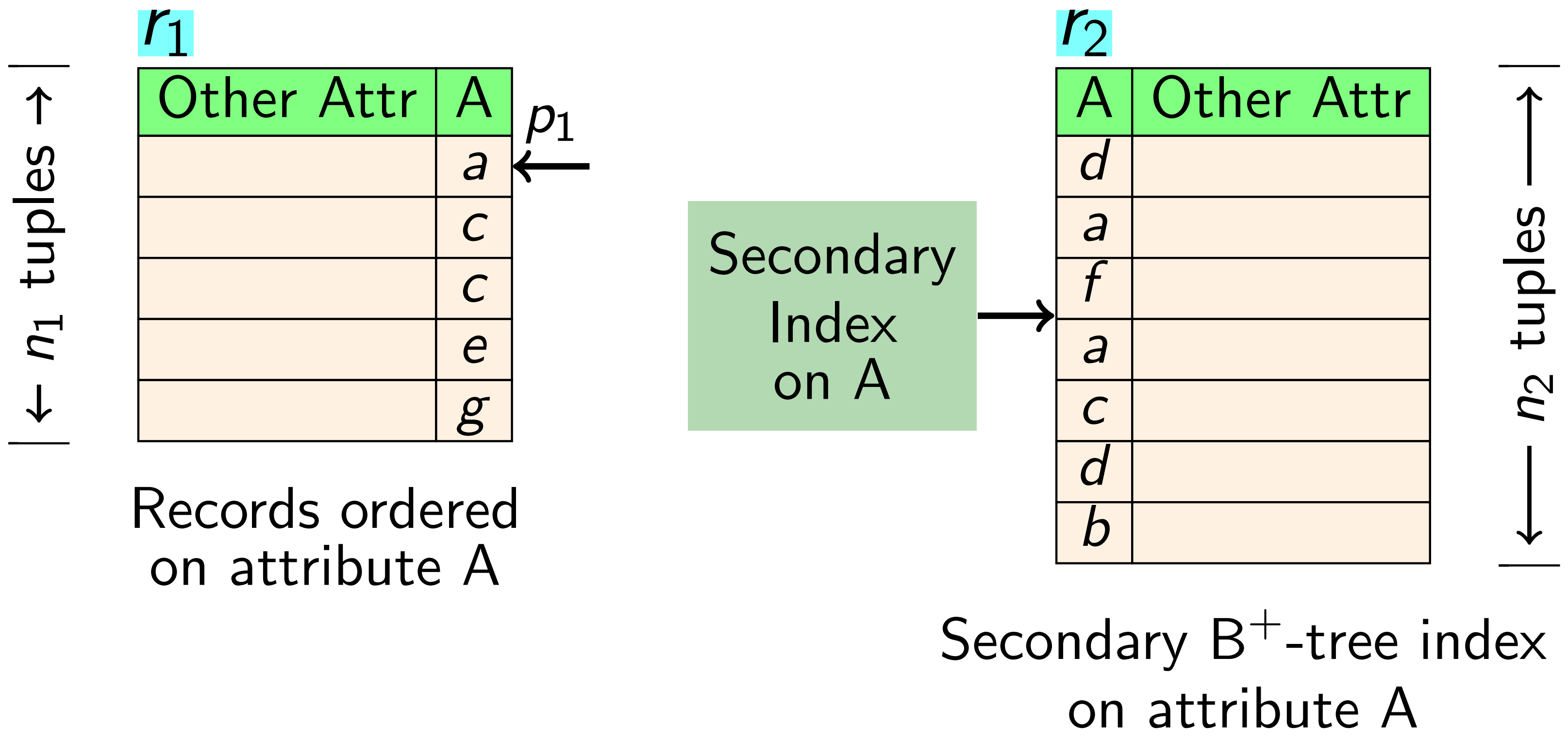
- Merge join is a very effective approach, but applies only when both relations have primary indices on the join attribute(s).
- The algorithm is similar to that of external merging, although the lists are not actually merged.
- Pointers are maintained to the sorted lists of tuples for each relation.
- The one pointing to the lesser value is advanced.
- Equal attribute values for the two pointers identifies a joinable pair.
- The complexity is $\Theta(n_1 + n_2)$, but the constant is likely to be much smaller than for the indexed join described previously.



r_1
Other Attr
A
a
c
c
e
g
p_1
$\leftarrow n_1$ tuples $\rightarrow$
r_2
A
Other Attr
b
c
d
e
e
f
h
p_2
$\leftarrow n_2$ tuples $\rightarrow$

Hybrid Merge Join

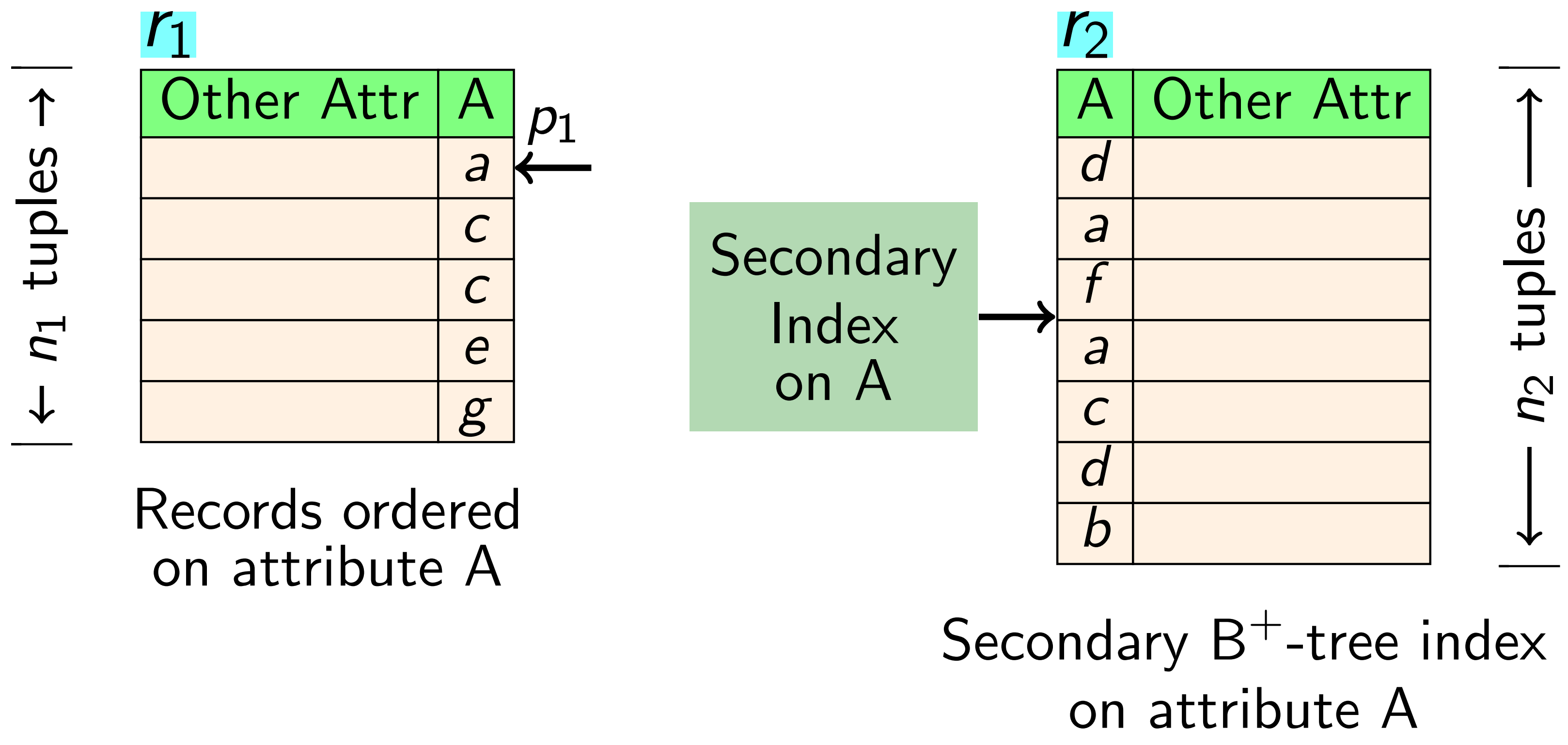
- Here the records of only one relation are sorted, but the other has a secondary (B^+-tree) index.
- The leaves of that B^+-tree are pairs of the form (v, p), with v the value of attribute A and p a pointer to the actual record.
    - They are sorted on v only.
- This gives effectively a sorted list of the records of r_2, but it is only a list of pointers, not of physical records.



Hybrid Merge Join — 2

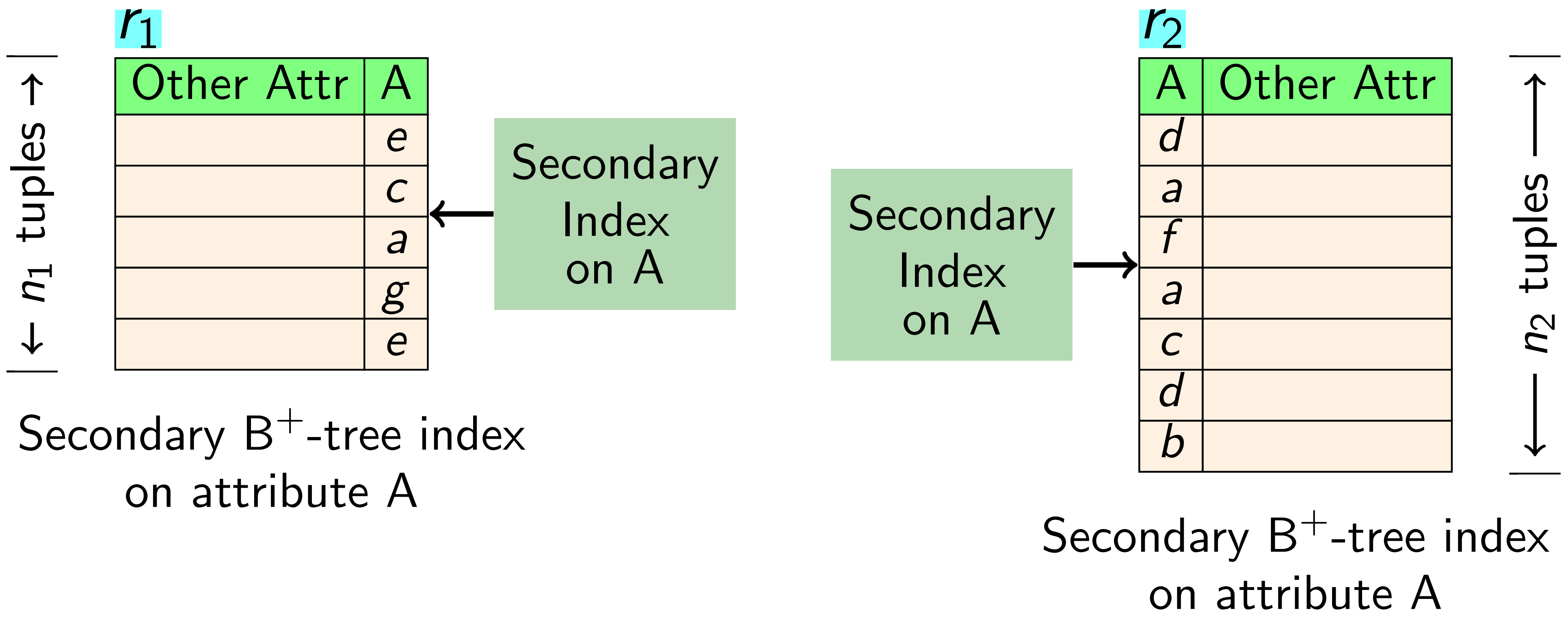
Recall: The leaves of that B$^+$-tree are pairs of the form (v, p), with v the value of attribute A and p a pointer to the actual record.

- The records of r_1 may be matched with the leaves of the B$^+$-tree of r_2 via a merge-like procedure, with sorting on the common attribute values.
- This merged list may be furthermore optimized by doing a secondary sort on the pointer values p for each group of entries from r_2 with the same value for the key v.



Double Hybrid Merge Join

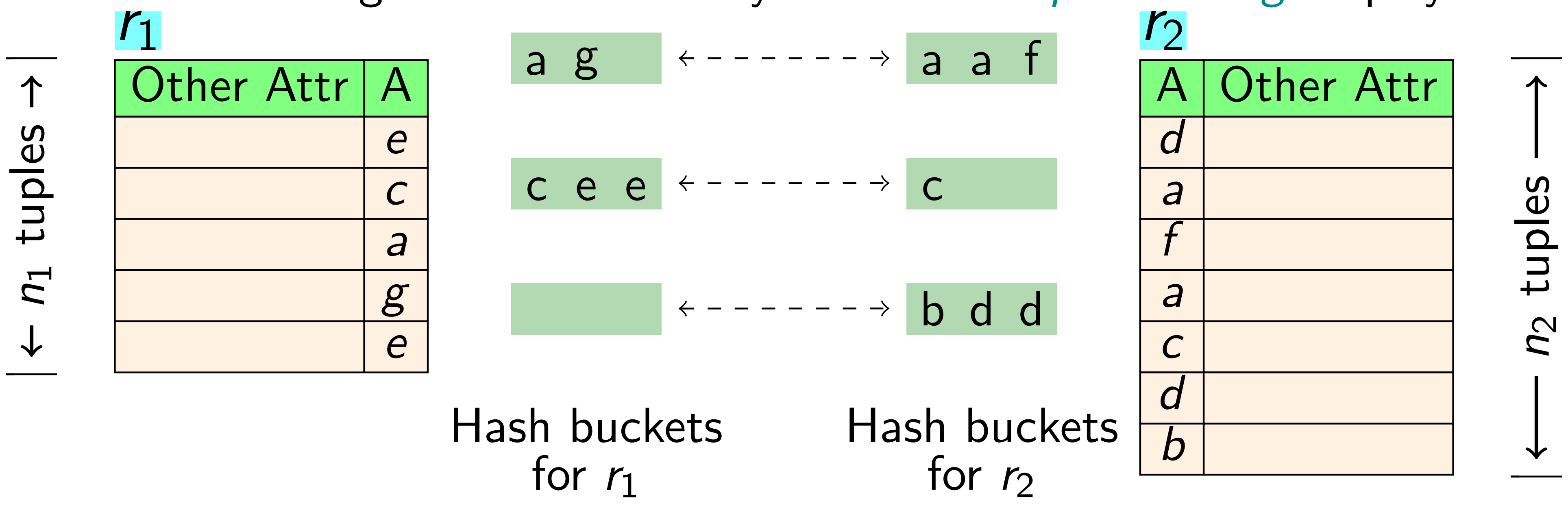
- This same approach may be applied in the case that both relations have secondary B$^+$-tree indices on the attributes to be joined.
- Here the matching is effectively only on pointers to the actual records.



r_1
Other Attr
A
e
c
a
g
e
$\leftarrow n_1$ tuples $\rightarrow$
Secondary Index on A
Secondary B$^+$-tree index on attribute A
r_2
A
Other Attr
d
a
f
a
c
d
b
$\leftarrow n_2$ tuples $\rightarrow$
Secondary Index on A
Secondary B$^+$-tree index on attribute A

GRACE Hash Join

- Hash join is widely used when no indices on join attributes are available.
- In GRACE hash join, for each relation, the values for the join attribute(s) are hashed into buckets using the same hash function.
- The buckets should be small enough so that each corresponding pair of buckets fits into main memory, but otherwise, the bigger the better.
- Matches for the join will always be found within the corresponding buckets for each relation.
- Usually, an index is built for each bucket to facilitate searching.
- Blocks are too large for main memory $\Rightarrow$ *recursive partitioning* employed.



Details and Analysis of GRACE Hash Join

- GRACE hash join is divided into two main steps.

Step 1: Build the buckets: Copy each tuple in each relation to the appropriate bucket.

Step 2: Compute the join: Find matching tuples in each bucket pair by *probing* the bucket of one relation (the *probe relation*) for each tuple of the matching bucket of the other relation (the *build relation*), and add those tuples to the resulting join.

Parameters: There are three parameters which will be used throughout the analysis.

n_{blk_i}: The number of disk blocks for relation r_i.

n_{buf}: The number of disk blocks which fit into the in-memory buffer.

n_{hash}: The number of hash buckets per relation.

GRACE Hash Join — Step 1

Data Structures for Step 1 — Build the buckets: For this step, the in-memory buffer is divided into two parts:

InBuf: Contains input records from the relation to be processed.

OutBuf: Contains records which will be placed in the hash buckets.

- OutBuf is partitioned into n_{hash} sub-buffers, one for each bucket.

Algorithm for Step 1:

- Process the relation sequentially, allocating each tuple to the sub-buffer for its bucket.
- When a sub-buffer becomes full, flush it to the true bucket in secondary memory.

Number of block transfers for Step 1: Each tuple is transferred twice, once to placed into InBuf and once to be written from OutBuf to its bucket.

$$2 \cdot (n_{\mathrm{blk}_1} + n_{\mathrm{blk}_2})$$

- May be slightly more due to partially filled blocks.

Details and Analysis of GRACE Hash Join — 2

Data Structures for Step 2: As in Step 1, there is InBuf and OutBuf.

InBuf: Contains the matching buckets for a given index.

OutBuf: Contains records which are in the join.

Algorithm for Step 2: The process is very simple:

- In turn, bring each pair of matching buckets into memory.
- Identify joinable tuples and write them to the result.

Number of block transfers for Step 2: Each tuple is transferred once.

$$n_{\mathrm{blk}_1} + n_{\mathrm{blk}_2}$$

- If the result must be written to secondary storage, additional transfers will be required.

Total block transfers for both steps:

$$3 \cdot (n_{\mathrm{blk}_1} + n_{\mathrm{blk}_2})$$

Further Aspects of GRACE Hash Join

- Not every block transfer requires a disk seek.
- It is therefore useful to compute the number of disk seeks.

Additional parameters:

n_{InBuf}: The number of disk blocks in InBuf.

n_{OutBuf}: The number of disk blocks in OutBuf.

Disk seeks for Step 1:

$$\left\lceil \frac{n_{\text{blk}_1}}{n_{\text{InBuf}}} \right\rceil + \left\lceil \frac{n_{\text{blk}_2}}{n_{\text{InBuf}}} \right\rceil + \left\lceil \frac{n_{\text{blk}_1} \times n_{\text{hash}}}{n_{\text{OutBuf}}} \right\rceil + \left\lceil \frac{n_{\text{blk}_2} \times n_{\text{hash}}}{n_{\text{OutBuf}}} \right\rceil$$

Disk seeks for Step 2:

$$\left\lceil \frac{n_{\text{blk}_1}}{n_{\text{InBuf}}} \right\rceil + \left\lceil \frac{n_{\text{blk}_2}}{n_{\text{InBuf}}} \right\rceil$$

Total disk seeks for both steps:

$$2 \cdot \left(\left\lceil \frac{n_{\text{blk}_1}}{n_{\text{InBuf}}} \right\rceil + \left\lceil \frac{n_{\text{blk}_2}}{n_{\text{InBuf}}} \right\rceil \right) + \left\lceil \frac{n_{\text{blk}_1} \times n_{\text{hash}}}{n_{\text{OutBuf}}} \right\rceil + \left\lceil \frac{n_{\text{blk}_2} \times n_{\text{hash}}}{n_{\text{OutBuf}}} \right\rceil$$

Recursive Partitioning for GRACE Hash Join

- If $n_{\text{hash}} > n_{\text{OutBuf}}$, that is, if n_{hash} is larger than the number of blocks available in the OutBuf, it is not possible to apply the algorithm directly.
- Rather, *recursive partitioning* is necessary.
- In this case, the maximum number of hash buckets which will fit in OutBuf is used in the first step.
- Then each such bucket is brought into main memory individually and further subdivided in a second step.
- This process is repeated until the desired number of buckets is reached.
- Recursive partitioning adds a factor which is log in the number of blocks.
- With the larger memory sizes which are common nowadays, recursive partitioning is seldom required.

Hybrid Hash Join

- There is a simple extension to GRACE hash join which is widely used when primary memory is plentiful.
- Rather than allocate the entire memory buffer to InBuf and OutBuf, part is allocated to store some of the buckets of the probe relation.
- The idea is then to construct as much of the join as possible in main memory, thus avoiding intermediate steps which require disk access.
- Assume that there are k buckets in total for each relation, with $\{B_{i1}, B_{i2}, \ldots, B_{ik}\}$ denoting the buckets for r_i, $i \in \{1, 2\}$.
- Suppose that there is enough additional memory to store the first ℓ buckets of the probe relation (assumed to be r_1) in main memory.
- Then these buckets are never written to disk. Rather, the members of $\{B_{11}, \ldots, B_{1\ell}\}$ are kept in the main-memory buffer.
- When processing the build relation r_2, construct directly the part of the join corresponding to matches on $\{B_{11}, \ldots, B_{1\ell}\}$ as the tuples of r_2 are processed, thus avoiding the construction of $\{B_{21}, \ldots, B_{2\ell}\}$ completely.
- This will result in substantial savings on disk I/O.

Algorithms for Projection

- There is little of a special nature which can be done for projection.
- Each record must be processed in turn, discarding the undesired attributes.
- In the case that the retained attributes do not form a key, it may be necessary to remove duplicates (if the query requires it).

Algorithms for Removal of Duplicates

- There are two basic ways to remove duplicates from a list.

Sort and scan: The list is first sorted on a key, and then processed sequentially, with all but the first occurrence of the key removed.

  - The (N-way) merge sort algorithm is used, and governs the complexity.

On-the-fly index creation: The list is processed sequentially, building an index of all keys which have occurred.

  - If the key which is found in a given step is already present in the index, the containing tuple is discarded.
  - If the index would be too large for main memory, hashing (as in hash join) can be used, with the duplicate-removal process applied to each bucket.
  - Complexity analysis is similar to that for hash join.

- In general, the removal of duplicates is an expensive operation, and so must be requested explicitly.

Set Operations

- The binary set operations include *union* ($\cup$), *intersection* ($\cap$), and *difference* ($\setminus$).
- For intersection and difference, records with identical key values must be found in order to eliminate tuples which occur in one of the operands but not in the result.
  - Duplicates do not arise for intersection and difference.
- For union, removal of duplicates is not always required, but when it is, records with identical key values must be found in order to find the tuples which occur more than once.
- In short, for intersection and difference, as well as for union with duplicates removed, records with identical key values must be found in each of the two sets.
- To find these matching values, the main options are sorting and hashing.
- Each of these approaches will be described in turn.

Set Operations via Sorting

Set intersection using sorting: The most straightforward approach is to sort the lists first, and then use an approach similar to that for merge join to identify the matching elements.

Set union using sorting: The most straightforward approach is to combine the two sets of records.

- If elimination of duplicates is required, sort the result, discarding duplicates.
- If the two input sets are already sorted, then a procedure similar to merge join may be used, but this time an element is kept if it occurs in either list.

Set difference using sorting: The approach is similar to that for intersection, except that a tuple from the first list is discarded rather than kept if a matching tuple is found in the second list.

- Unlike union and intersection, this operation is not symmetric in the two lists.

- In all cases, the computation of complexity is straightforward, based upon previous analyses.

Set Operations via Hashing

- Suppose that a set operation on relations r_1 and r_2 is to be performed.
- The first step in each case is to construct matching sets of hash buckets $\{r_{i1}, r_{i2}, \dots, r_{ik}\}$ for $i \in \{1, 2\}$, as in the hash join.
- Next, for each j, $1 \leq j \leq k$, do the following:
  - Bring r_{1j} and r_{2j} into main memory.
  - Build an index of r_{1j}.
  - Complete the step listed below.

Set intersection using hashing: To compute $r_{1j} \cap r_{2j}$, for each tuple in r_{2i}, probe the index of r_{1i} and delete the tuple from r_1 if no match is found.

Set difference using hashing: To compute $r_{1j} \setminus r_{2j}$, for each tuple in r_{2i}, probe the index of r_{1i} and delete the tuple from r_1 if a match is found.

Set union using hashing: To compute $r_{1j} \cup r_{2j}$, for each tuple in r_{2i}, probe the index of r_{1i} and add that tuple to r_1 is no match is found.

- The results for each j are then combined.
- In all cases, the computation of complexity is straightforward, based upon previous analyses.
- A hybrid approach, along the lines of hybrid hash join, is also possible.

Aggregation

Example: Average salary by department:

```
SELECT dept_name, avg(salary) FROM instructor GROUP BY dept_name;
```

- The difficult part, with respect to computational complexity, is to partition the tuples into the groups.
- The approach for partitioning is similar to that for join, and involves sorting and/or hashing.

Two main approaches:

First group, then aggregate: Compute the aggregate group (group by `dept_name` in the example), and then perform the aggregation (averaging in the example).

Aggregate on the fly, as groups are computed: As elements are added to each group, update the aggregation.

- This works directly for operations such as `count`, `sum`, `min`, and `max`.
- For `avg`, `count` and `sum` are aggregated on the fly, and `avg` is computed from them at the end.