

Query Processing

COSC 404 – Database System Implementation



Query Processing Overview



The goal of the query processor is very simple:

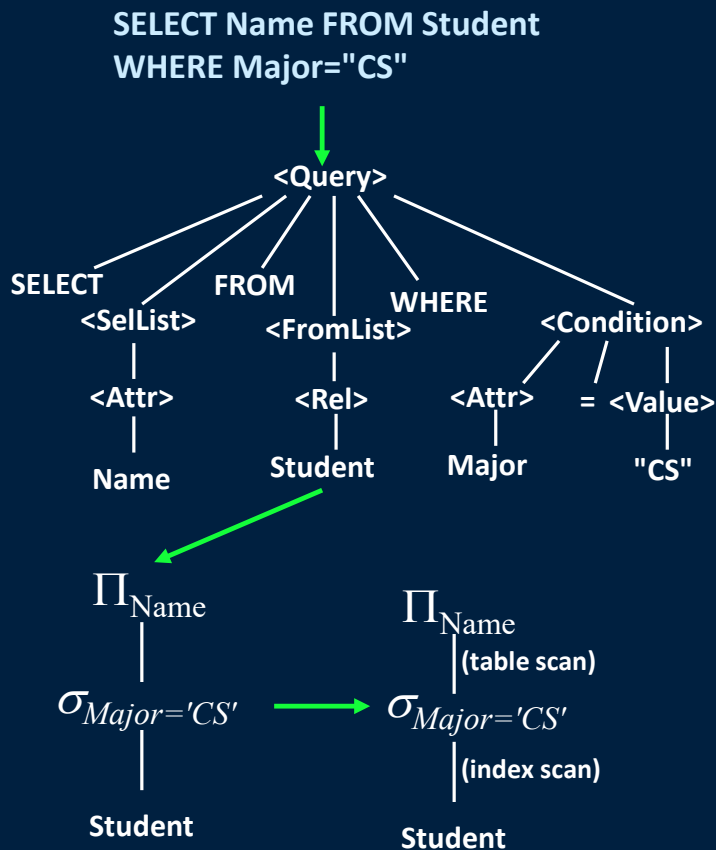
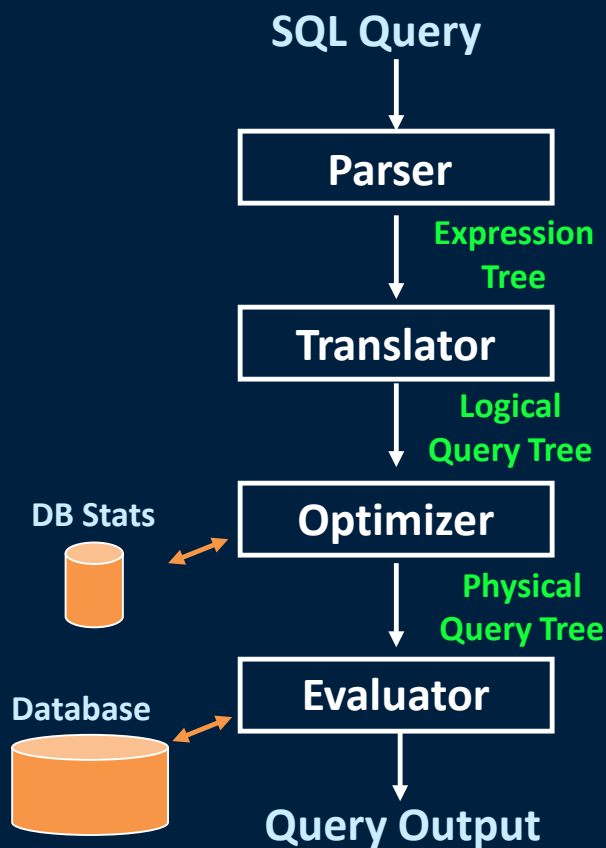
Return the answer to a SQL query in the most efficient way possible given the organization of the database.

Achieving this goal is anything but simple:

- Different file organizations and indexing affect performance.
- Different algorithms can be used to perform the relational algebra operations with varying performance based on the DB.
- Estimating the cost of the query itself is hard.
- Determining the best way to answer one query in isolation is challenging. How about many concurrent queries?



Components of a Query Processor



Review: SQL Query Summary

The general form of the `SELECT` statement is:

```
SELECT <attribute list>  
FROM   <table list>  
[WHERE (condition)]  
[GROUP BY <grouping attributes>]  
[HAVING <group condition>]  
[ORDER BY <attribute list>]
```

- Clauses in square brackets ([,]) are optional.
- There are often numerous ways to express the same query in SQL.

Review: SQL and Relational Algebra

The `SELECT` statement can be mapped directly to relational algebra.

```
SELECT  $A_1, A_2, \dots, A_n$   
FROM    $R_1, R_2, \dots, R_m$   
WHERE   $P$ 
```

is equivalent to:

$$\Pi_{A_1, A_2, \dots, A_n}(\sigma_P(R_1 \times R_2 \times \dots \times R_m))$$

Review: Relational Algebra Operators

Relational Operators:

- selection σ - return subset of rows
- projection π - return subset of columns
- Cartesian product \times - all combinations of two relations
- join \bowtie - combines σ and \times
- duplicate elimination δ - eliminates duplicates

Set operators:

- Union \cup - tuple in output if in either or both
- Difference $-$ - tuple in output if in 1st but not 2nd
- Intersection \cap - tuple in output if in both
- Union compatibility means relations must have the same number of columns with compatible domains.

Review: Selection and Projection

The **selection operation** returns an output relation that has a subset of the tuples of the input by using a **predicate**.

The **projection operation** returns an output relation that contains a subset of the attributes of the input.

Note: Duplicate tuples are eliminated.

Input Emp Relation

eno	ename	title	salary
E1	J. Doe	EE	30000
E2	M. Smith	SA	50000
E3	A. Lee	ME	40000
E4	J. Miller	PR	20000
E5	B. Casey	SA	50000
E6	L. Chu	EE	30000
E7	R. Davis	ME	40000
E8	J. Jones	SA	50000

Selection Example

$\sigma_{\text{salary} > 35000 \text{ OR title} = 'PR'}(\text{Emp})$

eno	ename	title	salary
E2	M. Smith	SA	50000
E3	A. Lee	ME	40000
E4	J. Miller	PR	20000
E5	B. Casey	SA	50000
E7	R. Davis	ME	40000
E8	J. Jones	SA	50000

Projection Example

$\Pi_{\text{eno}, \text{ename}}(\text{Emp})$

eno	ename
E1	J. Doe
E2	M. Smith
E3	A. Lee
E4	J. Miller
E5	B. Casey
E6	L. Chu
E7	R. Davis
E8	J. Jones

Review: Cartesian Product

The **Cartesian (or cross) product** of two relations R (of degree k_1) and S (of degree k_2) combines the tuples of R and S in all possible ways.

The result of $R \times S$ is a relation of degree $(k_1 + k_2)$ and consists of all $(k_1 + k_2)$ -tuples where each tuple is a concatenation of one tuple of R with one tuple of S . The cardinality of $R \times S$ is $|R| * |S|$.

Emp Relation

eno	ename	title	salary
E1	J. Doe	EE	30000
E2	M. Smith	SA	50000
E3	A. Lee	ME	40000
E4	J. Miller	PR	20000

Proj Relation

pno	pname	budget
P1	Instruments	150000
P2	DB Develop	135000
P3	CAD/CAM	250000

Emp \times Proj

eno	ename	title	salary	pno	pname	budget
E1	J. Doe	EE	30000	P1	Instruments	150000
E2	M. Smith	SA	50000	P1	Instruments	150000
E3	A. Lee	ME	40000	P1	Instruments	150000
E4	J. Miller	PR	20000	P1	Instruments	150000
E1	J. Doe	EE	30000	P2	DB Develop	135000
E2	M. Smith	SA	50000	P2	DB Develop	135000
E3	A. Lee	ME	40000	P2	DB Develop	135000
E4	J. Miller	PR	20000	P2	DB Develop	135000
E1	J. Doe	EE	30000	P3	CAD/CAM	250000
E2	M. Smith	SA	50000	P3	CAD/CAM	250000
E3	A. Lee	ME	40000	P3	CAD/CAM	250000
E4	J. Miller	PR	20000	P3	CAD/CAM	250000

Review: Join

Theta (θ) join combines cross product and selection: $R \bowtie_F S = \sigma_F(R \times S)$.

An **equijoin** only contains the equality operator (=) in the join predicate.

◆ e.g. $\text{WorksOn} \bowtie_{\text{WorksOn.pno} = \text{Proj.pno}} \text{Proj}$

A **natural join** $R \bowtie S$ is the equijoin of R and S over a set of attributes common to both R and S that removes duplicate join attributes.

WorksOn Relation

eno	pno	resp	dur
E1	P1	Manager	12
E2	P1	Analyst	24
E2	P2	Analyst	6
E3	P4	Engineer	48
E5	P2	Manager	24
E6	P4	Manager	48
E7	P3	Engineer	36
E7	P4	Engineer	23

Proj Relation

pno	pname	budget
P1	Instruments	150000
P2	DB Develop	135000
P3	CAD/CAM	250000
P4	Maintenance	310000
P5	CAD/CAM	500000

$\text{WorksOn} \bowtie_{\text{WorksOn.pno} = \text{Proj.pno}} \text{Proj}$

eno	pno	resp	dur	P.pno	pname	budget
E1	P1	Manager	12	P1	Instruments	150000
E2	P1	Analyst	24	P1	Instruments	150000
E2	P2	Analyst	6	P2	DB Develop	135000
E3	P4	Engineer	48	P4	Maintenance	310000
E5	P2	Manager	24	P2	DB Develop	135000
E6	P4	Manager	48	P4	Maintenance	310000
E7	P3	Engineer	36	P3	CAD/CAM	250000
E7	P4	Engineer	23	P4	Maintenance	310000

Review Question

Given this table and the query:

```
SELECT eno, salary
FROM emp
WHERE salary >= 40000
```

How many rows in the result?

- A) 2
- B) 3
- C) 4
- D) 5

Emp Relation

<u>eno</u>	ename	title	salary
E1	J. Doe	EE	30000
E2	M. Smith	SA	50000
E3	A. Lee	ME	40000
E4	J. Miller	PR	20000
E5	B. Casey	SA	50000
E6	L. Chu	EE	30000
E7	R. Davis	ME	40000
E8	J. Jones	SA	50000

Review Question

Given these tables and the query:

$$\Pi_{eno, ename} (\sigma_{title='EE'} (Emp \bowtie_{emp.dno=dept.dno} Dept))$$

Dept Relation

<u>dno</u>	dname	mgreno
D1	Management	E8
D2	Consulting	E7
D3	Accounting	E5
D4	Development	null

How many rows in the result?

Emp Relation

<u>eno</u>	ename	bdate	title	salary	supereno	dno
E1	J. Doe	01-05-75	EE	30000	E2	null
E2	M. Smith	06-04-66	SA	50000	E5	D3
E3	A. Lee	07-05-66	ME	40000	E7	D2
E4	J. Miller	09-01-50	PR	20000	E6	D3
E5	B. Casey	12-25-71	SA	50000	E8	D3
E6	L. Chu	11-30-65	EE	30000	E7	D2
E7	R. Davis	09-08-77	ME	40000	E8	D1
E8	J. Jones	10-11-72	SA	50000	null	D1

- A) 0
- B) 1
- C) 2
- D) 8

Review Question

Question: What is the symbol for duplicate elimination?

A) σ

B) \times

C) π

D) \bowtie

E) δ

Algorithms for Relational Operators

Our initial focus is developing algorithms to implement the relational operators of selection, projection, and join.

The query processor contains these implementations and uses them to answer queries.

We will discuss when the algorithms should be applied when discussing optimization. For now, we will build a *toolkit* of potential algorithms that could be used.

Query Processing

Classifying Algorithms



Two ways to classify relational algebra algorithms:

1) By the number of times the data is read:

- **One-Pass** - selection or projection operators or binary operators where one relation fits entirely in memory.
- **Two-Pass** - data does not fit entirely in memory in one pass, but algorithm can process data using only two passes.
- **Multi-Pass** - generalization to larger data sets.

2) By the type of relational algebra operator performed:

- **Tuple-at-a-time, unary operators** - selection, projection
 - Do not need entire relation to perform operation.
- **Full-relation, unary operators** - grouping, duplicate elimination
- **Full-relation, binary operators** - join, set operations

Measuring Cost of Algorithms

Algorithms will be compared using number of block I/Os.

- Note: CPU time is important but requires more detailed models.

Assumptions:

- The arguments of any operator are found on disk, but the operator result is left in memory.
 - For example, a select operation on a relation, must read the relation from disk, but after the operation is performed, the result is left in memory (and can be potentially used by the next operator).
 - This is also true for the query result.

Measuring Cost of Algorithms (2)

Some basic statistics will be useful when discussing algorithms:

- 1) The number of buffer blocks available to the algorithm is M .
 - We will assume memory blocks are the same size as disk blocks.
 - The buffers are used to store input and intermediate results; the buffers do not have to be used to store output which is assumed to go elsewhere.
 - M is always less than the size of memory, but in practice, may even be much smaller than that as many operators can be executing at once.
- 2) $B(R)$ or just B (if R is assumed) is the # of blocks on disk used to store all the tuples of R .
 - Usually, assume that R is clustered and that we can only read 1 block at a time. Note that we will ignore free-space in blocks even though in practice blocks are not normally kept completely full.
- 3) $T(R)$ or just T (if R is assumed) is the # of tuples in R .
- 4) $V(R, a)$ is the # of distinct values of the column a in R .
 - Note: $V(\text{Student}, Id) = T(\text{Student})$ as Id is a key.

Metrics Question

Question: The number of rows in table `Emp` is 50. There are 10 possible values for the `title` attribute. Select a true statement.

- A) $T(\text{Emp}) = 10$
- B) $V(\text{Emp}, \text{eno}) = 10$
- C) $V(\text{Emp}, \text{title}) = 10$
- D) $V(\text{Emp}, \text{title}) = 50$

Scans and Sorts

Two basic operations are scanning and sorting an input.

There are two types of scans:

- 1) **Table scan** - read the relation R from disk one block at a time.
- 2) **Index scan** - read the relation R or only the tuples of R that satisfy a given condition, by using an index on R .

Sorting can be performed in three ways:

- 1) **Index sort** - used when the relation R has a B+-tree index on sort attribute a .
- 2) **Main-memory sort** - read the entire relation R into main memory and use an efficient sorting algorithm.
- 3) **External-merge sort** - use the external-merge sort if the entire relation R is too large to fit in memory.
 - We will discuss this sorting algorithm later.

Measuring Cost of Scan Operators

The cost of a table scan for relation R is B .

What would be the cost of an index scan of relation R that has B data blocks and I index blocks?

- Does it depend on the type of index?

Iterators for Operators

Database operations are implemented as *iterators*.

- Also called *pipelining* or *producer-consumer*.

Instead of completing the entire operation before releasing output, an operator releases output to other operators as it is produced *one tuple at a time*.

Iterators are combined into a tree of operators. Iterators execute in parallel and query results are produced faster.



Structure of Iterators

Database iterators implement three methods:

- `init()` - initialize the iterator variables and algorithm.
 - Starts the process, but does not retrieve a tuple.
- `next()` - return the next tuple of the result and perform any required processing to be able to return the next tuple of the result the next time `next()` is called.
 - `next()` returns `NULL` if there are no more tuples to return.
- `close()` - destroy iterator variables and terminate the algorithm.

Each algorithm we discuss can be implemented as an iterator.

Iterator Example

Table Scan Iterator



```
init() {
    b = the first block of R;
    t = first tuple of R;
}

next() {
    if (t is past the last tuple on block b) {
        increment b to the next block;
        if (there is no next block)
            return NULL;
        else /* b is a new block */
            t = first tuple on block b;
    }
    oldt = t;
    increment t to the next tuple of b;
    return oldt;
}

close() {}
```


Iterator Example

Main-Memory Sort Iterator



```
init() {  
    Allocate buffer array A  
    read entire relation R block-by-block into A;  
    sort A using quick sort;  
    tLoc = 0;    // First tuple location in A  
}
```

```
next() {  
    if (tLoc >= T)  
        return NULL;  
    else  
    {  
        tLoc++;  
        return A[tLoc-1];  
    }  
}
```

```
close() {}
```

How is this iterator different than the table scan iterator?

Programming Iterators in Java

We will implement iterators in Java and combine them to build execution trees.

Iterators are derived from the `Operator` class.

This class has the methods `init()`, `next()`, `hasNext()`, and `close()`.

The operator has an array of input operators which may consist of 0, 1, or 2 operators.

- A relation scan has 0 input operators.

Operator class

```
public abstract class Operator
{
    protected Operator[] input;           // Input operators
    protected int numInputs;              // # of inputs
    protected Relation outputRelation;    // Output relation
    protected int BUFFER_SIZE;            // # of buffer pages
    protected int BLOCKING_FACTOR;        // # of tuples per page

    Operator()                            {this(null, 0, 0); }
    Operator(Operator []in, int bfr, int bs) { ... }
    // Iterator methods
    abstract public void init() throws IOException;

    abstract public Tuple next() throws IOException;

    public void close() throws IOException
    {
        for (int i=0; i < numInputs; i++)
            input[i].close();
    }
}
```

Scan Operator Example

```
public class TextFileScan extends Operator
{
    protected String inFileName;           // Name of input file to scan
    protected BufferedReader inFile;       // Reader for input file
    protected Relation inputRelation;      // Schema of file scanned

    public TextFileScan(String inName, Relation r)
    {
        super(); inFileName = inName;
        inputRelation = r; setOutputRelation(r);
    }

    public void init() throws FileNotFoundException, IOException
    {
        inFile = FileManager.openTextInputFile(inFileName);
    }

    public Tuple next() throws IOException
    {
        Tuple t = new Tuple(inputRelation);
        if (!t.readText(inFile))           // Read a tuple from input file
            return null;
        return t;
    }

    public void close() throws IOException
    {
        FileManager.closeFile(inFile);
    }
}
```

Sort Operator Example

```
public class Sort extends Operator
{
    public Sort(Operator in, SortComparator sorter)
    {
        // Initializes local variables ...
    }
    public void init() throws IOException, FileNotFoundException
    {
        input[0].init();
        buffer = new Tuple[arraySize];           // Initialize buffer
        int count = 0;
        while (count < arraySize)
        {
            if ( (buffer[count] = input[0].next()) == null)
                break;
            count++;
        }
        curTuple = 0;
        Arrays.sort(buffer, 0, count, sorter);
        input[0].close();
    }
    public Tuple next() throws IOException
    {
        if (curTuple < arraySize)
            return buffer[curTuple++];
        return null;
    }
    // Note: close() method is empty
}
```

Projection Operator Example

```
public class Projection extends Operator
{
    protected ProjectionList plist;                // Projection information

    public Projection(Operator in, ProjectionList pl)
    {
        super(new Operator[] {in}, 0, 0);
        plist = pl;
    }
    public void init() throws IOException
    {
        input[0].init();
        Relation inR = input[0].getOutputRelation();
        setOutputRelation(inR.projectRelation(plist));
    }

    public Tuple next() throws IOException
    {
        Tuple inTuple = input[0].next();
        if (inTuple == null)
            return null;
        return new Tuple(...perform projection using plist from inTuple);
    }
    public void close() throws IOException
    {
        super.close(); }
}
```

Answering Queries Using Iterators

Given the user query:

```
SELECT *  
FROM nation
```

This code would answer the query:

```
TextFileScan op = new TextFileScan("data/nation.txt", r);  
op.init();  
  
Tuple t;  
t = op.next();  
while (t != null)  
{  
    System.out.println(t);  
    t = op.next();  
}  
op.close();
```


Iterator Practice Questions

Write the code to answer the query:

```
SELECT  *
FROM    nation
ORDER BY n_name
```

- Assume that a `SortComparator` `sc` has been defined that you can pass in to the `Sort` object to sort appropriately.

Challenge: Answer this query:

```
SELECT  n_nationkey, n_name
FROM    nation
ORDER BY n_name
```

- Assume you can provide an array of attribute names to the `Projection` operator.

One-Pass Algorithms

One-pass algorithms read data from the input only once.

Selection and projection are one-pass, tuple-at-a-time operators.

Tuple-at-a-time operators require only one main memory buffer ($M=1$) and cost the same as the scan.

- Note that the CPU cost is the dominant cost of these operators.

One-Pass Algorithms

Grouping and Duplicate Elimination



Duplication elimination (δ) and grouping (γ) require reading the entire relation and remembering tuples previously seen.

One-pass duplicate elimination algorithm:

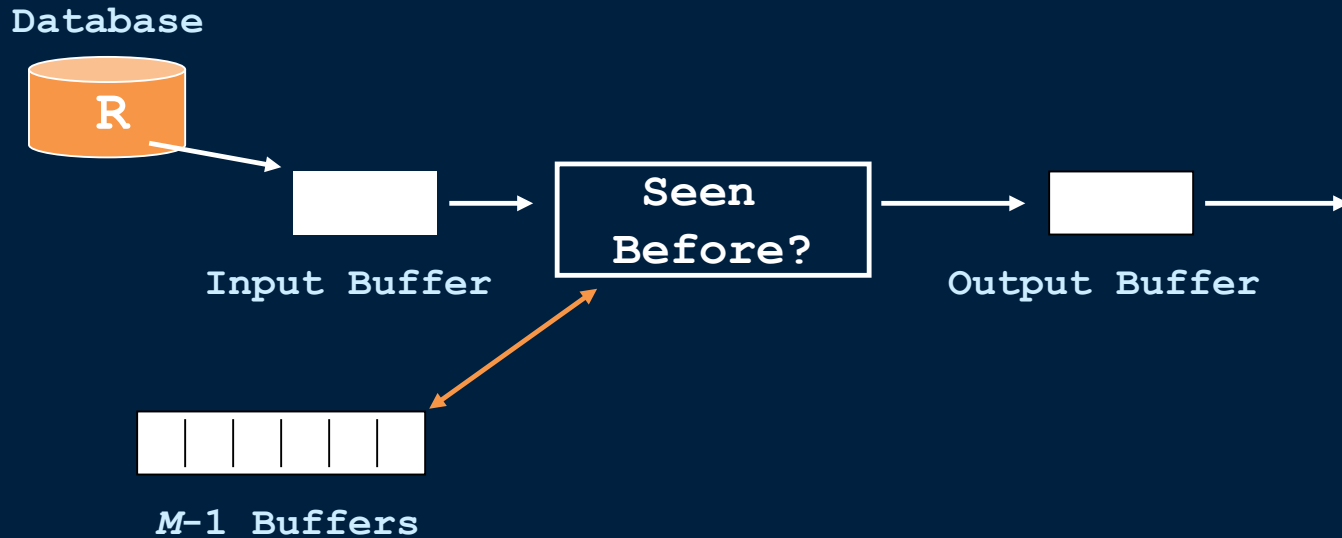
- 1) Read each block of relation R one at a time.
- 2) For each tuple read, determine if:
 - This is the first time the tuple has been seen. If so, copy to output.
 - Otherwise, discard duplicate tuple.

Challenge: How do we know if a tuple has been seen before?

Answer: We must build a main memory data structure that stores copies of all the tuples that we have already seen.

One-Pass Algorithms

Duplicate Elimination Overview



How do we use
these buffers?

One-Pass Algorithms

Duplicate Elimination Discussion



The **$M-1$** buffers are used to store a fast lookup structure such that given a tuple, we can determine if we have seen it before.

- Main-memory hashing or balanced binary trees are used.
 - Note that an array would be inefficient. Why?
- Space overhead for the data structure is ignored in our calculations.

$M-1$ buffers allows us to store **$M-1$** blocks of **R** . Thus, the number of main memory buffers required is approximately:

$$M \geq B(\delta(R))$$

One Pass Duplicate Elimination Question

Question: If $T(R)=100$ and $V(R,a)=1$ and we perform $\delta(\Pi_a(R))$, select a true statement.

- A)** The maximum memory size used is 100 tuples (not counting input tuple).
- B)** The size of the result is 100 tuples.
- C)** The size of the result is unknown.
- D)** The maximum memory size used is 1 tuple (not counting input tuple).

One-Pass Algorithms

Grouping



The grouping (γ) operator can be evaluated similar to duplicate elimination except now besides identifying if a particular group already exists, we must also calculate the aggregate values for each group as requested by the user.

How to calculate aggregate values:

- $\text{MIN}(a)$ or $\text{MAX}(a)$ - for each group maintain the minimum or maximum value of attribute a seen so far. Update as required.
- $\text{COUNT}(\ast)$ - add one for each tuple of the group seen.
- $\text{SUM}(a)$ - keep a running sum for a for each group.
- $\text{AVG}(a)$ - keep running sum and count for a for each group and return $\text{SUM}(a)/\text{COUNT}(a)$ after all tuples are seen.

One-Pass Algorithms

Grouping Example



```
SELECT Major, Count(*), Min(Year), Max(Year), AVG(Year)
FROM Student GROUP BY Major
```

Student Relation

St. ID	Name	Mjr	Yr
10567	J. Doe	CS	3
11589	T. Allen	BA	2
15973	M. Smith	CS	3
29579	B. Zimmer	BS	1
34596	T. Atkins	ME	4
75623	J. Wong	BA	3
84920	S. Allen	CS	4
96256	P. Wright	ME	2



Memory Buffers

Major	Count	Min	Max	Avg
CS	3	3	4	3.3
BA	2	2	3	2.5
BS	1	1	1	1
ME	2	2	4	3

Main memory table copied to output to answer query.

One-Pass Algorithms

Grouping Discussion



After all tuples are seen and aggregate values are calculated, write each tuple representing a group to the output.

The cost of the algorithm is $B(R)$, and the memory requirement M is almost always less than $B(R)$, although it can be much smaller depending on the group attributes.

- Question: When would M ever be larger than $B(R)$?

Both duplicate elimination and grouping are **blocking algorithms**.

One-Pass Algorithms

Binary Operations



It is also possible to implement one-pass algorithms for the binary operations of union, intersection, difference, cross-product, and natural join.

For the set operators, we must distinguish between the set and bag versions of the operators:

- **Union** - set union (\cup_S) and bag union (\cup_B)
- **Intersection** - set intersection (\cap_S) and bag intersection (\cap_B)
- **Difference** - set difference ($-_S$) and bag difference ($-_B$)

Note that only bag union is a tuple-at-a-time algorithm. All other operators require one of the two operands to fit entirely in main memory in order to support a one-pass algorithm.

- We will assume two operand relations **R** and **S**, with **S** being small enough to fit entirely in memory.

Binary Operations - General Algorithm

The general algorithm is similar for all binary operations:

- 1) Read the smaller relation, S , entirely into main memory and construct an efficient search structure for it.
 - This requires approximately $B(S)$ main memory blocks.
- 2) Allocate one buffer for reading one block of the larger relation, R , at a time.
- 3) For each block and each tuple of R
 - Compare the tuple of R with the tuples of S in memory and perform the specific function required for the operator.

The function performed in step #3 is operator dependent.

All binary one-pass algorithms take $B(R) + B(S)$ disk operations.

They work as long as $B(S) \leq M-1$ or $B(S) < M$.

One-Pass Algorithms

Binary Operations Algorithms



Function performed on each tuple t of R for the operators:

- 1) **Set Union** - If t is not in S , copy to output, otherwise discard.
 - Note: All tuples of S were initially copied to output.
- 2) **Set Intersection**-If t is in S , copy to output, otherwise discard.
 - Note: No tuples of S were initially copied to output.
- 3) **Set difference**
 - $R - S$: If t is not in S , copy to output, otherwise discard.
 - $S - R$: If t is in S , then delete t from the copy of S in main memory. If t is not in S , do nothing. After seeing all tuples of R , copy to output tuples of S that remain in memory.
- 4) **Bag Intersection**
 - Read S into memory and associate a count for each distinct tuple.
 - If t is found in S and count is still positive, decrement count by 1 and output t . Otherwise, discard t .

One-Pass Algorithms

Binary Operations Algorithms (2)



Function performed on each tuple t of R for the operators:

- 5) **Bag difference**
 - $S -_B R$: Similar to bag intersection (using counts), except only output tuples of S at the end if they have positive counts (and output that many).
 - $R -_B S$: Exercise - try it for yourself.
- 6) **Cross-product** - Concatenate t with each tuple of S in main memory. Output each tuple formed.
- 7) **Natural Join**
 - Assume connecting relations $R(X,Y)$ and $S(Y,Z)$ on attribute set Y .
 - X is all attributes of R not in Y , and Z is all attributes of S not in Y .
 - For each tuple t of R , find all tuples of S that match on Y .
 - For each match output a joined tuple.

One-Pass Algorithms

Review Questions



- 1) How many buffers are required to perform a selection operation on a relation that has size 10,000 blocks?
- 2) Assume the number of buffers $M=100$. Let $B(R)=10,000$ and $B(S)=90$. How many block reads are performed for $R \cup S$?
- 3) If $M=100$, $B(R)=5,000$ and $B(S)=1,000$, how many block reads are performed for $R - S$ using a one-pass algorithm?



Nested-Loop Joins

Nested-loop joins are join algorithms that compute a join using simple `for` loops.

These algorithms are essentially "one-and-a-half-pass" algorithms because one of the relations is read only once, while the other relation is read repeatedly.

There are two variants:

- 1) Tuple-based nested-loop join
- 2) Block-based nested-loop join

For this discussion, we will assume a natural join is to be computed on relations $R(X,Y)$ and $S(Y,Z)$.

Tuple-based Nested-Loop Join

In the tuple-based nested-loop join, tuples are matched using two `for` loops. Algorithm:

```

for (each tuple s in S)
    for (each tuple r in R)
        if (r and s join to make a tuple t)
            output t;
  
```

Notes:

- Very simple algorithm that can vary widely in performance if:
 - There is an index on the join attribute of **R**, so the entire relation **R** does not have to be read.
 - Memory is managed smartly so that tuples are in memory when needed (use buffers intelligently).
 - Worse case is $T(R) \cdot T(S)$ if for every tuple we have to read it from disk!

Tuple-based Nested-Loop Join Iterator

```
init() { R.init(); S.init(); s = S.next(); }  
next() {  
    do {  
        r = R.next();  
        if (r == NULL) { // R is exhausted for current s  
            R.close();  
            s = S.next();  
            if (s == NULL) return NULL; // Done  
            R.init(); // Re-initialize scan of R  
            r = R.next();  
        }  
    } while !(r and s join); // Found one joined tuple  
    return (the tuple created by joining r and s);  
}  
close() { R.close(); S.close(); }
```

Block-based Nested-Loop Join

Block-based nested-loop join is more efficient because it operates on blocks instead of individual tuples.

Two major improvements:

- 1) Access relations by blocks instead of by tuples.
- 2) Buffer as many blocks as available of the outer relation **S**. That is, load chunks of relation **S** into the buffer at a time.

The first improvement makes sure that as we read **R** in the inner loop, we do it a block at a time to minimize I/O.

The second improvement enables us to join one tuple of **R** (inner loop) with as many tuples of **S** that fit in memory at one time (outer loop).

- This means that we do not have to continually load a block of **S** at time.



Nested-Block Join Algorithm

```
for (each chunk of  $M-1$  blocks of  $S$ )  
    read these blocks into main memory buffers  
    organize these tuples into an efficient search  
        structure whose search key is the join attributes  
    for (each block  $b$  of  $R$ )  
        read  $b$  into main memory  
        for (each tuple  $t$  of  $b$ )  
            find tuples of  $S$  in memory that join with  $t$   
            output the join of  $t$  with each of these tuples
```

Note that this algorithm has 3 for loops, but does the same processing more efficiently than the tuple-based algorithm. Outer loop processes tuples of S , inner loop processes tuples of R .

★ Nested-Loop Joins

Analysis and Discussion



Nested-block join analysis:

Assume S is the smaller relation.

of outer loop iterations = $\lceil B(S)/M-1 \rceil$

Each iteration reads $M-1$ blocks of S and $B(R)$ (all) blocks of R .

Number of disk I/O is:

$$B(S) + B(R) * \left\lceil \frac{B(S)}{M-1} \right\rceil$$

In general, this can be approximated by $B(S) * B(R)/M$.

Nested-Loop Joins

Performance Example



If $M=1,000$, $B(R)=100,000$, $T(R)=1,000,000$, $B(S)=5,000$, and $T(S)=250,000$, calculate the performance of tuple-based and block-based nested loop joins.

Tuple-Based Join:

$$\begin{aligned}\text{worst case} &= T(R) * T(S) = 1,000,000 * 250,000 \\ &= \mathbf{25,000,000,000} = 25 \text{ billion!}\end{aligned}$$

Block-Based Join:

$$\begin{aligned}\text{worst case} &= B(S) + B(R) * \text{ceiling}(B(S)/(M-1)) \\ &= 5,000 + 100,000 * \text{ceiling}(5,000 / 999) \\ &= \mathbf{605,000}\end{aligned}$$

Question: Calculate the I/Os if the larger relation R is in the outer loop.

Block Nested Loop Join Question

Question: Select a true statement.

- A)** BNLJ buffers the smaller relation in memory.
- B)** BNLJ buffers the larger relation in memory.

Sorting-based Two-Pass Algorithms

Two-pass algorithms read the input at most twice.

Sorting-based two-pass algorithms rely on the external sort merge algorithm to accomplish their goals.

The basic process is as follows:

- 1) Create sorted sublists of size M blocks of the relation R .
- 2) Merge the sorted sublists by continually taking the minimum value in each list.
- 3) Apply the appropriate function to implement the operator.

We will first study the external sort-merge algorithm then demonstrate how its variations can be used to answer queries.



External Sort-Merge Algorithm

1) Create sorted runs as follows:

- Let i be 0 initially, and M be the number of main memory blocks.
- Repeat these steps until the end of the relation:
 - (a) Read M blocks of relation into memory.
 - (b) Sort the in-memory blocks.
 - (c) Write sorted data to run R_i ; increment i .

2) Merge the runs in a single merge step:

- Suppose for now that $i < M$. Use i blocks of memory to buffer input runs.
- We will write output to disk instead of using 1 block to buffer output.
- Repeat these steps until all input buffer pages are empty:
 - (a) Select the first record in sorted order from each of the buffers.
 - (b) Write the record to the output.
 - (c) Delete the record from the buffer page. If the buffer page is empty, read the next block (if any) of the run (sublist) into the buffer.

External Sort-Merge Example

Sort by column #1. $M=3$. (Note: Not using an output buffer.)

initial relation

G	24
A	19
D	31
C	33
B	14
E	16
R	6
D	21
M	3

Runs

A	19	✓
D	31	✓
G	24	✓
B	14	✓
C	33	✓
E	16	✓
D	21	✓
M	3	✓
R	6	✓

sorted relation

A	19
B	14
C	33
D	21
D	31
E	16
G	24
M	3
R	6

Merge
Pass

Create Sorted
Sublists

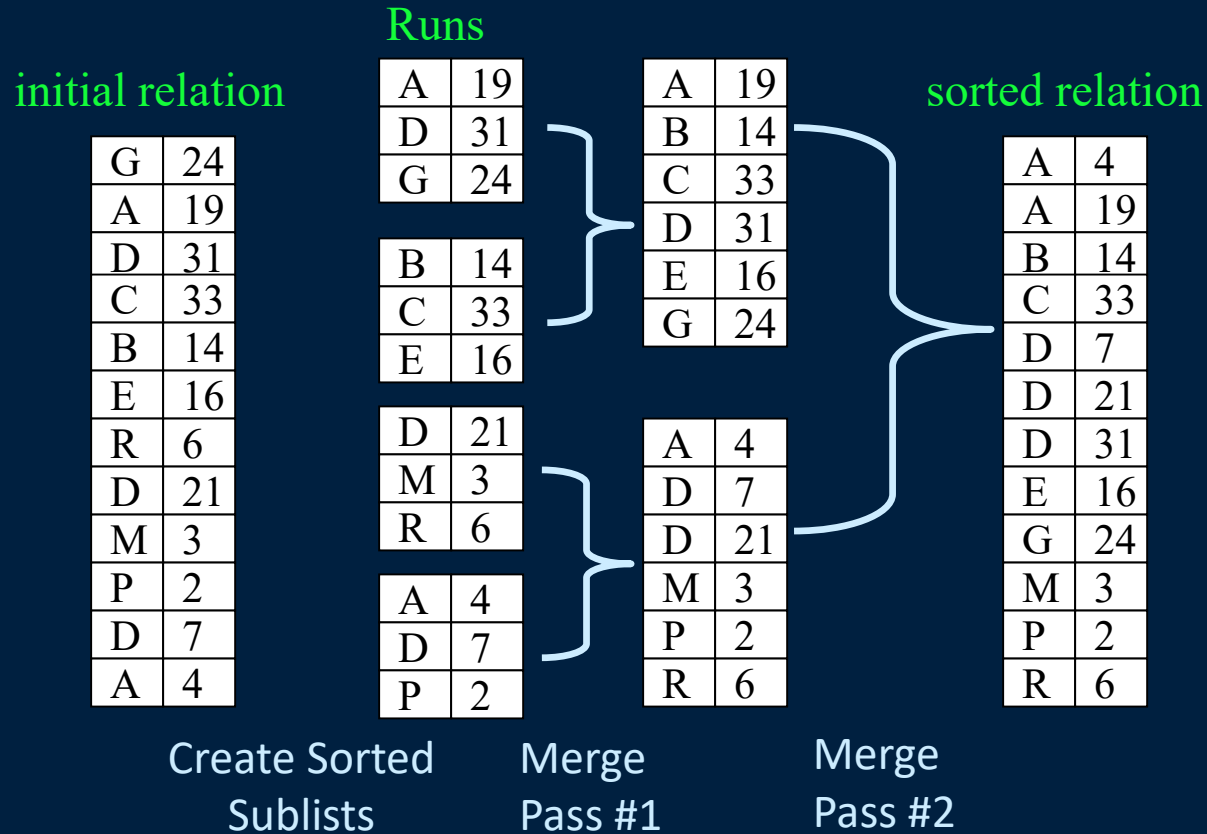
Multi-Pass External Sort-Merge

If $i \geq M$, several merge passes are required as we can not buffer the first block of all sublists in memory at the same time.

- In this case, use an output block to store the result of a merge.
- In each pass, contiguous groups of $M-1$ runs are merged.
- A pass reduces the number of runs by a factor of $M-1$, and creates runs longer by the same factor.
- Repeated passes are performed until all runs are merged.

External Sort-Merge Example 2

Multi-Pass Merge



External Sort-Merge Analysis

Cost analysis:

- Two-pass external sort cost is: $3*B$. ($B=B(R)$)
 - Each block is read twice: once for initial sort, once for merge.
 - Each block is written once after the first pass.
 - The cost is $4*B$ if we include the cost of writing the output.
- Multi-pass external sort cost is: $B*(2\lceil \log_{M-1}(B/M) \rceil + 1)$.
 - Disk accesses for initial run creation as well as in each pass is $2*B$ (except for final pass that does not write out results).
 - Total number of merge passes required: $\lceil \log_{M-1}(B/M) \rceil$
 - B/M is the # of initial runs, and # decreases by factor of $M-1$ every pass.
 - Each pass reads/writes each block ($2*B$) except final run has no write.

Sort analysis:

- A two-pass external sort can sort M^2 blocks.
- A N -pass external sort can sort M^N blocks.

External Sort-Merge Analysis Example

A main memory size is 64 MB, the block size is 4 KB, and the record size is 160 bytes.

- 1) How many records can be sorted using a two-pass sort?
 - Sort can sort M^2 memory blocks.
 - # of memory blocks = memory size/block size
 - Total # of blocks sorted = $(64 \text{ MB} / 4 \text{ KB})^2 = \text{approx. } 268 \text{ million}$
 - Total # of records sorted = #blocks *blockingFactor = approx. 6.8 billion!
 - Total size is approximately **1 terabyte**.
- 2) How many records can be sorted using a three-pass sort?
 - Sort can sort M^3 memory blocks.
 - Same calculation results in 112 trillion records of total size **16 petabytes!**

Bottom-line: Two way sort is sufficient for most purposes!

External Sort-Merge Usage

The external sort-merge algorithm can be used when:

- 1) SQL queries specify a sorted output
- 2) For processing a join algorithm using merge-join algorithm.
- 3) Duplicate elimination
- 4) Grouping and aggregation
- 5) Set operations

We will see how the basic external sort-merge algorithm can be modified for these operations.

Duplicate Elimination Using Sorting

Algorithm (two-pass):

- Sort the tuples of R into sublists using the available memory buffers M .
- In the second pass, buffer one block of each sublist in memory like the sorting algorithm.
- However, in this case, instead of sorting the tuples, only copy one to output and ignore all tuples with duplicate values.
 - Every time we copy one value to the output, we search forward in all sublists removing all copies of this value.

Duplicate Elimination Example

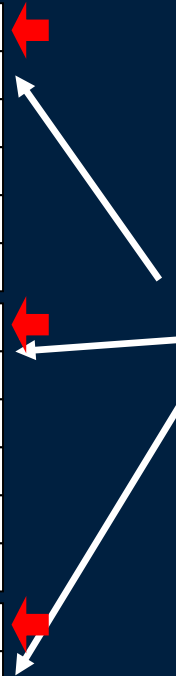
initial relation

2	
5	
2	
1	
2	
2	
4	
5	
4	
3	
4	
2	
1	
5	
2	



Runs

1	
2	
2	
2	
2	
5	
2	
3	
4	
4	
4	
4	
5	
1	
2	
5	



First blocks (each with 2 records) are initially loaded into memory.

Duplicate elimination on column #1. $M=3$. blocking factor=2.

Duplicate Elimination Example (2)

initial relation

2	
5	
2	
1	
2	
2	
4	
5	
4	
3	
4	
2	
1	
5	
2	

Runs

2	
2	
2	
2	
5	
2	
3	
4	
4	
4	
5	
2	
5	

output result

1	
---	--

Duplicate Elimination Example (3)

initial relation

2	
5	
2	
1	
2	
2	
4	
5	
4	
3	
4	
2	
1	
5	
2	

Runs

5	
3	
4	
4	
4	
5	
5	

output result

1	
2	

Load new block.

Load new block.



Load new block.



Duplicate Elimination Example (4)

initial relation

2	
5	
2	
1	
2	
2	
4	
5	
4	
3	
4	
2	
1	
5	
2	

Runs

output result

1	
2	
3	
4	
5	

Final result.

Duplicate Elimination Analysis

The number of disk operations is always $3*B(R)$.

- $2*B(R)$ to read/write each block to create sorted sublists.
- $B(R)$ to read each block of each sublist when performing duplicate elimination.

Remember the single pass algorithm was $B(R)$.

The two-pass algorithm can handle relations where $B(R) \leq M^2$.

Grouping and Aggregation Using Sorting



Algorithm (two-pass):

- Sort the tuples of R into sublists using the available memory buffers M .
- In the second pass, buffer one block of each sublist in memory like the sorting algorithm.
- Find the smallest value of the sort key (grouping attributes) in all the sublists. This value becomes the next group.
 - Prepare to calculate all aggregates for this group.
 - Examine all tuples with the given value for the sort key and calculate aggregate functions accordingly.
 - Read blocks from the sublists into buffers as required.
 - When there are no more values for the given sort key, output a tuple containing the grouped values and the calculated aggregate values.

Analysis: This algorithm also performs $3*B(R)$ disk operations.

Grouping Question

initial relation

2	
5	
2	
1	
2	
2	
4	
5	
4	
3	
4	
2	
1	
5	
2	

Calculate the output for a query that groups by the given integer attribute and returns a count of the # of records that contains that attribute.

Assume $M=3$ and blocking factor=2.

Set Operations Using Sorting

The set operations can also be implemented using a sorting based algorithm.

- All algorithms start with an initial sublist creation step where both relations R and S are divided into sorted sublists.
- Use one main memory buffer for each sublist of R and S .
- Many of the algorithms require counting the # of tuples of R and S that are identical to the current minimum tuple t .

Special steps for each algorithm operation:

- **Set Union** - Find smallest tuple t of all buffers, copy t to output, and remove all other copies of t .
- **Set Intersection** - Find smallest tuple t of all buffers, copy t to output if it appears in both R and S .
- **Bag Intersection** - Find smallest tuple t of all buffers, output t the minimum # of times it appears in R and S .

Set Operations Using Sorting (2)

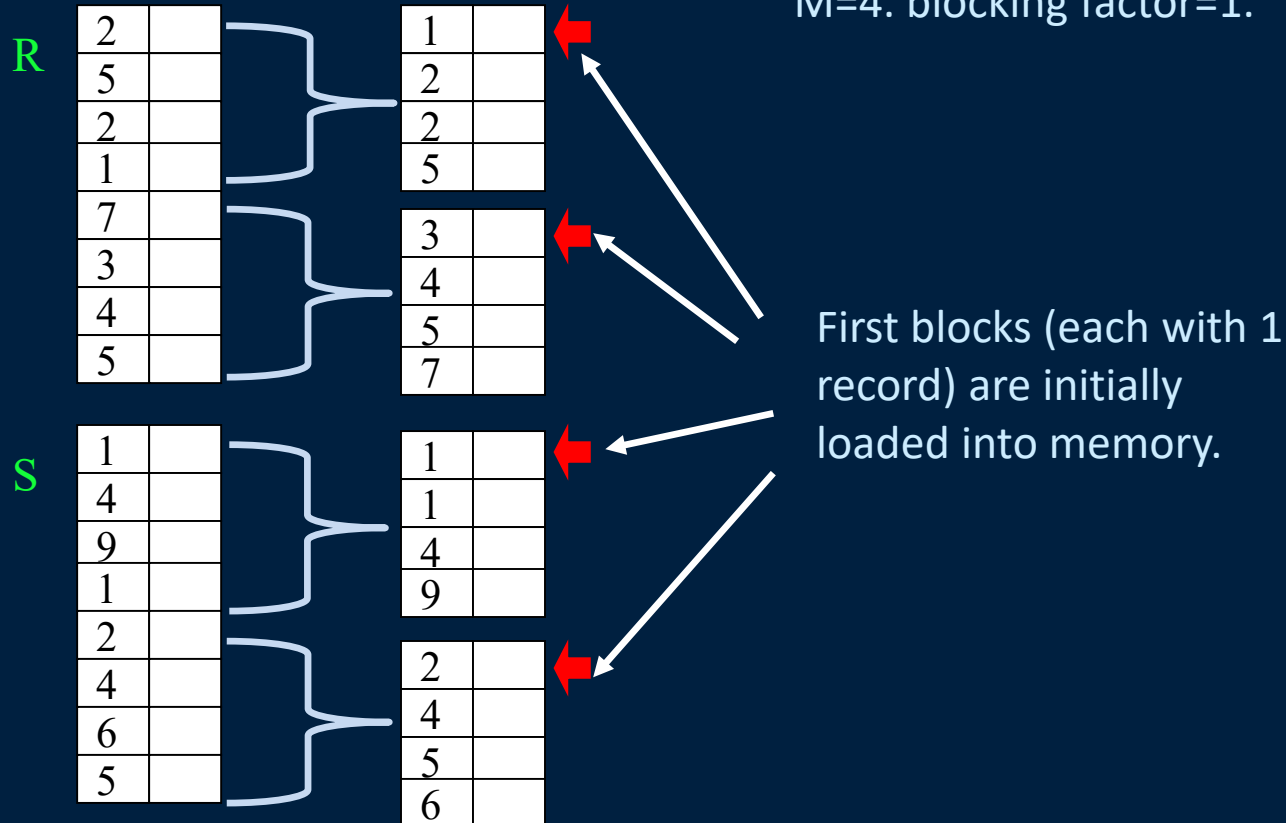
- **Set Difference** - Find smallest tuple t of all buffers, output t only if it appears in R but not in S . ($R -_S S$).
- **Bag difference** - Find smallest tuple t of all buffers, output t the number of times it appears in R minus the number of times it appears in S .

Analysis: All algorithms for set operations perform $3*(B(R)+B(S))$ disk operations, and two-pass versions only work if $B(R)+B(S) \leq M^2$.

- Note: More precisely the two-pass set algorithms only work if:

$$\lceil B(R)/M \rceil + \lceil B(S)/M \rceil \leq M$$

Set Operations Example - Intersection



Set Operations Example - Intersection (2)

Runs

2	
2	
5	



R

3	
4	
5	
7	



S

4	
9	



2	
4	
5	
6	



1 occurs in both R and S.

Output

1	
---	--

Output

72

Set Operations Questions

R

2	
5	
2	
1	
7	
3	
4	
5	

Show how the following operations are performed using two-pass sorting based algorithms:

- 1) Set Union
- 2) Set Difference ($R -_S S$)
- 3) Bag Difference
- 4) Bag Intersection

S

1	
4	
9	
1	
2	
4	
6	
5	

Assume $M=5$ and $bfr=1$.

For set operators, first eliminate duplicates in R and S.

Sort-Based Join Algorithm

Sorting can be used to join two relations $R(X,Y)$ and $S(Y,Z)$.

One of the challenges of any join algorithm is that the number of tuples of the two relations that share a common value of the join attribute(s) must be in memory at the same time.

- This is difficult if the number exceeds the size of memory.
 - Worse-case: Only one value for the join attribute(s). All tuples join to each other. If this is the case, block nested-loop join is used.

We will look at two different algorithms based on sorting:

- **Sort-join** - Allows for the most possible buffers for joining.
- **Sort-merge-join** - Has fewer I/Os, but more sensitive to large numbers of tuples with common join attribute.

Sort-Join Algorithm

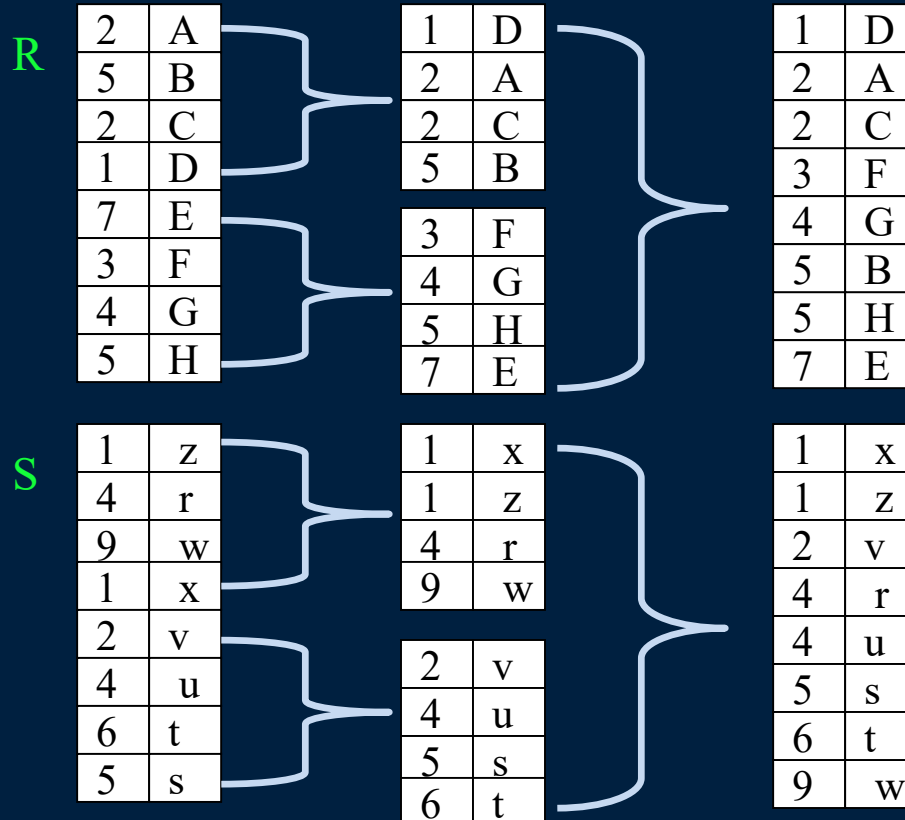
- 1) Sort R and S using an external merge sort with Y as the key.
- 2) Merge the sorted R and S using one buffer for each relation.
 - a) Find the smallest value y of join attributes Y in the start of blocks for R and S .
 - b) If y does not appear in the other relation, remove the tuples with key y .
 - c) Otherwise, identify all tuples in both relations that have the value y .
 - May need to read many blocks from R and S into memory. Use the M main memory buffers for this purpose.
 - d) Output all tuples that can be formed by joining tuples of R and S with common value y .
 - e) If either relation has no tuples buffered in memory, read the next block of the relation into a memory buffer.

Sort-Join Example

Sort Phase



M=4. blocking factor=1.



Sort-Join Example



Merge Phase



M=4. blocking factor=1.




R

1	D
2	A
2	C
3	F
4	G
5	B
5	H
7	E

 In memory after join on 1.


S

1	x
1	z
2	v
4	r
4	u
5	s
6	t
9	w

 Brought in for join on 1.

 In memory after join on 1.

Buffer

1	D
1	x
1	z

R
 S
 extra
 extra

Output

1	D	x
1	D	z

Notes:

- Only one block of R and S in memory at a time.
- Use other two buffers to bring in records with attribute values that match current join attribute.

Sort-Join Example

Merge Phase (2)



M=4. blocking factor=1.

R

1	D
2	A
2	C
3	F
4	G
5	B
5	H
7	E

← Brought in for join on 2.
← In memory after join on 2.

S

1	x
1	z
2	v
4	r
4	u
5	s
6	t
9	w

← In memory after join on 2.

Buffer

2	A	R
2	v	S
2	C	extra
		extra

Output

1	D	x
1	D	z
2	A	v
2	C	v

Sort-Join Example



Merge Phase (3)



M=4. blocking factor=1.




R

1	D
2	A
2	C
3	F
4	G
5	B
5	H
7	E



 In memory after join on 4.

S

1	x
1	z
2	v
4	r
4	u
5	s
6	t
9	w




 Brought in for join on 4.
 In memory after join on 4.

Buffer

4	G
4	r
4	u

R
 S
 extra
 extra

Output

1	D	x
1	D	z
2	A	v
2	C	v
4	G	r
4	G	u

Note: Skipped 3 in R because no match in S.

Sort-Join Example



Merge Phase (4)



M=4. blocking factor=1.


R

1	D
2	A
2	C
3	F
4	G
5	B
5	H
7	E

 Brought in for join on 5.
 In memory after join on 5.

S

1	x
1	z
2	v
4	r
4	u
5	s
6	t
9	w

 In memory after join on 5.

Buffer

5	B	R
5	s	S
5	H	extra
		extra

Output

1	D	x
1	D	z
2	A	v
2	C	v
4	G	r
4	G	u
5	B	s
5	H	s

Done as 7 (R) and 6,9 (S) do not match.

Sort-Join Analysis

The sort-join algorithm performs $5*(B(R)+B(S))$ disk operations.

- $4*B(R)+4*B(S)$ to perform the external merge sort on relations.
 - Counting the cost to output relations after sort - hence, $4*B$ not $3*B$.
- $1*B(R)+1*B(S)$ as each block of each relation read in merge phase to perform join.
- Algorithm limited to relations where $B(R) \leq M^2$ and $B(S) \leq M^2$.

The algorithm can use all the main memory buffers M to merge tuples with the same key value.

- If more tuples exist with the same key value than can fit in memory, then we could perform a nested-loop join just on the tuples with that given key value.
 - Also possible to do a one-pass join if the tuples with the key value for one relation all fit in memory.

Sort-Based Join Algorithm

Algorithm #1 - Example



Let relations R and S occupy 6,000 and 3,000 blocks respectively. Let $M = 101$ blocks.

Simple sort-join algorithm cost:

$$= 5 * (B(R) + B(S)) = \mathbf{45,000 \text{ disk I/Os}}$$

- Algorithm works because $6,000 \leq 101^2$ and $3,000 \leq 101^2$.
- Requires that there is no join value y where the total # of tuples from R and S with value y occupies more than 101 blocks.

Block nested-loop join cost:

$$= B(S) + B(S) * B(R) / (M - 1) = 183,000 \text{ (S as smaller relation)}$$

$$= B(S) + B(S) * B(R) / (M - 1) = 186,000 \text{ (S as larger relation)}$$

or approximately **180,000 disk I/Os**

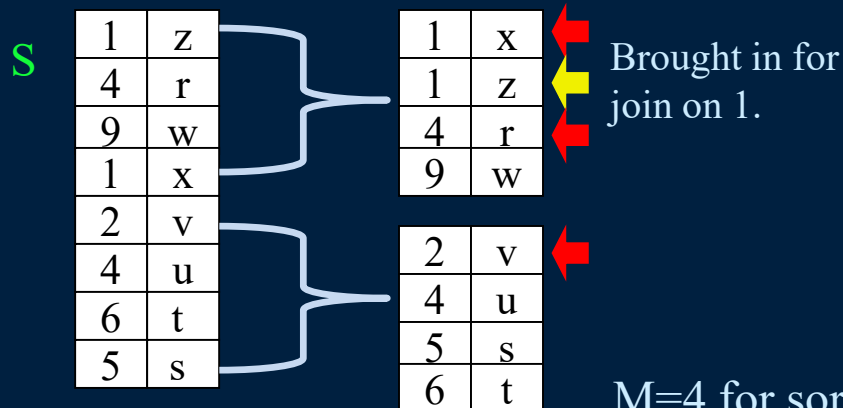
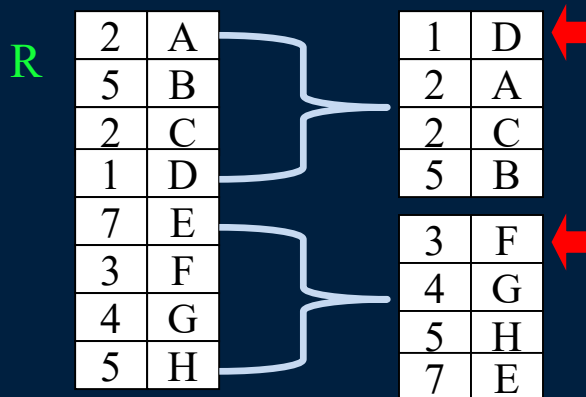
Sort-Merge-Join Algorithm

Idea: Merge the sorting steps and join steps to save disk I/Os.

Algorithm:

- 1) Create sorted sublists of size M using Y as the sort key for both R and S .
- 2) Buffer first block of all sublists in memory.
 - Assumes no more than M sublists in total.
- 3) Find the smallest value y of attribute(s) Y in all sublists.
- 4) Identify all tuples in R and S with value y .
 - May be able to buffer some of them if currently using less than M buffers.
- 5) Output the join of all tuples of R and S that share value y .

Sort-Merge-Join Example



Buffer

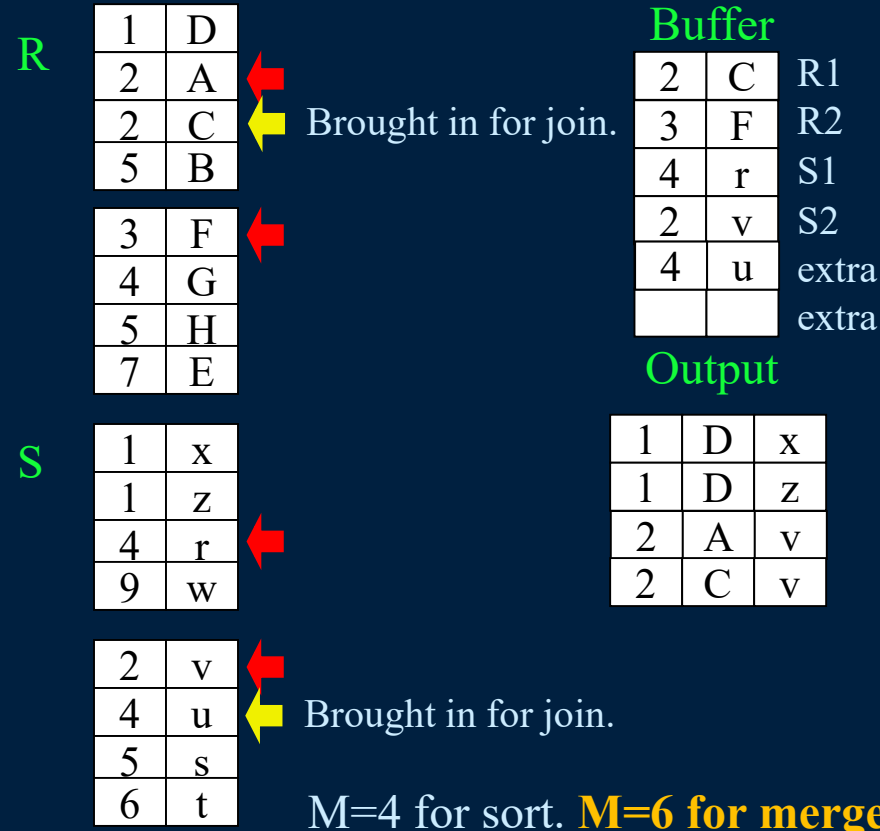
1	D	R1
3	F	R2
1	x	S1
2	v	S2
1	z	extra
4	r	extra

Output

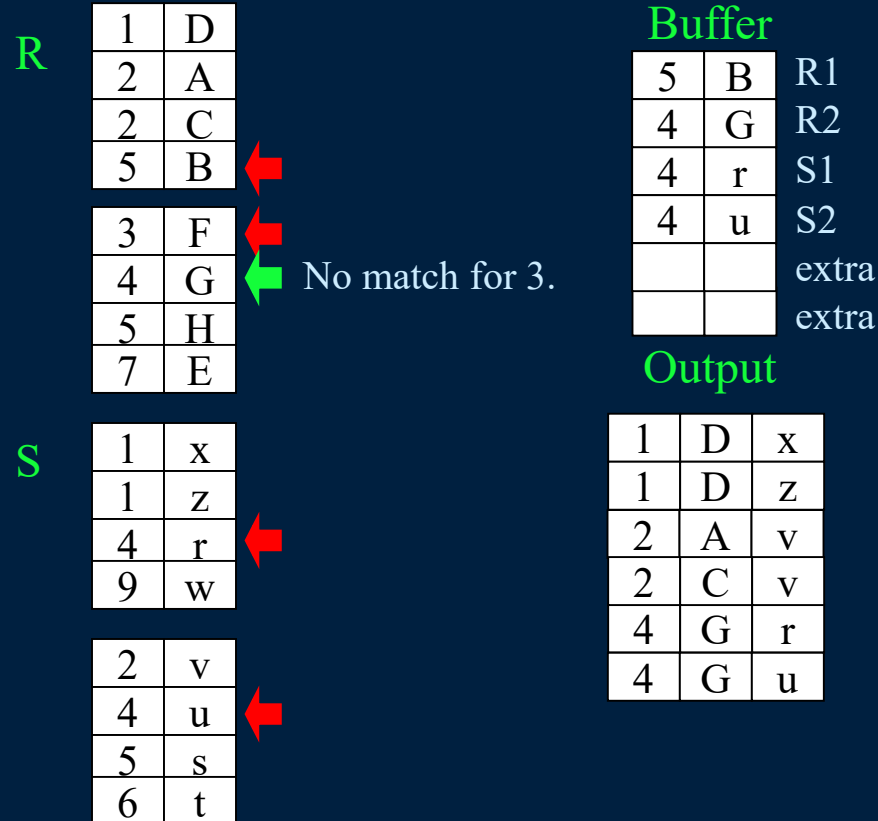
1	D	x
1	D	z

M=4 for sort. **M=6 for merge.**

Sort-Merge-Join Example (2)



Sort-Merge-Join Example (3)



Sort-Merge-Join Example (4)

R

1	D
2	A
2	C
5	B



3	F
4	G
5	H
7	E



S

1	x
1	z
4	r
9	w



2	v
4	u
5	s
6	t



Buffer

5	B
5	H
9	w
5	s

R1

R2

S1

S2

extra

extra

Output

1	D	x
1	D	z
2	A	v
2	C	v
4	G	r
4	G	u
5	B	s
5	H	s

Sort-Merge-Join Example (5)

Done!

R

1	D
2	A
2	C
5	B



3	F
4	G
5	H
7	E



No match for 7.

S

1	x
1	z
4	r
9	w



No match for 9.

2	v
4	u
5	s
6	t



No match for 6.

Buffer

7	E
9	w
6	t

R1

R2

S1

S2

extra

extra

Output

1	D	x
1	D	z
2	A	v
2	C	v
4	G	r
4	G	u
5	B	s
5	H	s

Sort-Merge-Join Analysis

Sort-merge-join algorithm performs $3*(B(R)+B(S))$ disk I/Os.

- $2*B(R)+2*B(S)$ to create the sublists for each relation.
- $1*B(R)+1*B(S)$ as each block of each relation read in merge phase to perform join.

The algorithm is limited to relations where $B(R)+B(S) \leq M^2$.

Sort-Merge-Join Example

Let relations R and S occupy 6,000 and 3,000 blocks respectively. Let $M = 101$ blocks.

Merge-sort-join algorithm cost:

$$= 3 * (B(R) + B(S)) = \mathbf{27,000 \text{ disk I/Os}}$$

- Algorithm works because $6,000 + 3,000 \leq 101^2$.
- # of memory blocks for sublists = 90
- 11 blocks free to use where there exists multiple join records with same key value y .



Summary of Sorting Based Methods

Performance of sorting based methods:

Operators	Approximate M required	Disk I/Os
γ, δ	\sqrt{B}	$3 * B$
$\cup, -, \cap$	$\sqrt{B(R) + B(S)}$	$3 * (B(R) + B(S))$
\bowtie (sort)	$\sqrt{\max(B(R), B(S))}$	$5 * (B(R) + B(S))$
\bowtie (sort-merge)	$\sqrt{B(R) + B(S)}$	$3 * (B(R) + B(S))$

Hashing-based Two-Pass Algorithms

Hashing-based two-pass algorithms use a hash function to group all tuples with the same key in the same bucket.

The basic process is as follows:

- 1) Use a hash function on each tuple to hash the tuple using a key to a **bucket** (or **partition**).
- 2) Perform the required operation by working on one bucket at a time. If there are M buffers available, $M-1$ is the number of buckets.

We start with the general external hash partitioning algorithm.



Partitioning Using Hashing Algorithm

- 1) Partition relation R using M buffers into $M-1$ buckets of roughly equal size.
- 2) Use a buffer for the input, and one buffer for each of the $M-1$ buckets.
- 3) When a tuple of relation R is read, it is hashed using the hash function $h(x)$ and stored in the appropriate bucket.
- 4) As output buffers (for the buckets) are filled they are written to disk. As the input buffer for R is exhausted, a new block is read.

The cost of the algorithm is $2*B(R)$.

Partitioning using Hashing Example

$M=4$, $bfr=3$, $h(x) = x \% 3$ (Hash on column #2.)

Buffers

Blocks

initial relation

(in memory)

(on disk)

G	24
A	19
D	31
C	33
B	14
E	16
R	6
D	21
M	3

input

G	24
A	19
D	31

$h(x) = 0$

G	24

$h(x) = 1$

A	19
D	31

$h(x) = 2$

Partitioning using Hashing Example (2)

$M=4$, $bfr=3$, $h(x) = x \% 3$ (Hash on column #2.)

Buffers

Blocks

initial relation

(in memory) (on disk)

G	24
A	19
D	31
C	33
B	14
E	16
R	6
D	21
M	3

input

C	33
B	14
E	16

Second input block

$h(x) = 0$

G	24
C	33

$h(x) = 1$

A	19
D	31
E	16

Save full block to disk.

$h(x) = 2$

B	14

Partitioning using Hashing Example (3)

$M=4$, $bfr=3$, $h(x) = x \% 3$ (Hash on column #2.)

Buffers

Blocks

initial relation

(in memory)

(on disk)

G	24
A	19
D	31
C	33
B	14
E	16
R	6
D	21
M	3

input

R	6
D	21
M	3

Third input block

$h(x) = 0$

G	24
C	33
R	6

$h(x) = 1$

A	19
D	31
E	16

$h(x) = 2$

B	14

Partitioning using Hashing Example (4)

$M=4$, $bfr=3$, $h(x) = x \% 3$ (Hash on column #2.)

Buffers

Blocks

initial relation

(in memory)

(on disk)

G	24
A	19
D	31
C	33
B	14
E	16
R	6
D	21
M	3

input

R	6
D	21
M	3

Third input block

$h(x) = 0$

D	21
M	3

G	24
C	33
R	6

$h(x) = 1$

A	19
D	31
E	16

$h(x) = 2$

B	14

Duplicate Elimination Using Hashing

Algorithm (two-pass):

- Partition tuples of R using hashing and $M-1$ buckets.
- Two copies of the same tuple will hash to the same bucket.
- One-pass algorithm can be used on each bucket to eliminate duplicates by loading entire bucket into memory.

Analysis:

- If all buckets are approximately the same size, each bucket R_i will be of size $B(R)/(M-1)$.
- The two-pass algorithm will work if $B(R) \leq M * (M-1)$.
- The # of disk operations is the same as for sorting, $3 * B(R)$.

Grouping and Aggregation Using Hashing



Algorithm (two-pass):

- Partition tuples of R using hashing and $M-1$ buckets.
- The hash function should **ONLY** use the grouping attributes.
- Tuples with the same values of the grouping attributes will hash to the same bucket.
- A one-pass algorithm is used on each bucket to perform grouping/aggregation by loading the entire bucket into memory.

The two-pass algorithm will work if **$B(R) \leq M * (M-1)$** .

On the second pass, we only need store one record per group.

- Thus, even if a bucket size is larger than M , we may be able to process it if all the group records in the bucket fit into M buffers.

The number of disk operations is **$3 * B(R)$** .

Grouping using Hashing Question

initial relation

2	
5	
2	
1	
2	
2	
4	
5	
4	
3	
4	
2	
1	
5	
2	

Calculate the output for a query that groups by the given integer attribute and returns a count of the # of records that contains that attribute.

Assume $M=4$ and blocking factor=2.

Set Operations Using Hashing

Set operations can be done using a hash-based algorithm.

- Start by hash partitioning R and S into $M-1$ buckets.
- Perform a one-pass algorithm for the set operation on each of the buckets produced.

All algorithms perform $3*(B(R) + B(S))$ disk operations.

Algorithms require that $\min(B(R), B(S)) \leq M^2$, since one of the operands must fit in memory after partitioning into buckets in order to perform the one-pass algorithm.

Hash Partitioning Question

Question: Given M memory buffers, how many hash buckets are used when hash partitioning?

- A) 1
- B) $M - 1$
- C) M
- D) $M + 1$

Hash-Join Algorithm

Hashing can be used to join two relations $R(X,Y)$ and $S(Y,Z)$.

Algorithm:

- Hash partition R and S using the hash key Y .
- If any tuple t_R of R will join with a tuple t_S of S , then t_R will be in bucket R_i and, t_S will be in bucket S_i . (same bucket index)
- For each bucket pair i , load the smaller bucket R_i or S_i into memory and perform a one-pass join.

Important notes for hash-based joins:

- The smaller relation is called the **build relation**, and the other relation is the **probe relation**. We will assume S is smaller.
- The size of the smaller relation dictates the number of partitioning steps needed.

Hash Join Example

Partition Phase



R

2	A
5	B
2	C
1	D
7	E
3	F
4	G
5	H

Partitions for R

$$h(x) = 0$$

3	F

$$h(x) = 1$$

1	D	4	G
7	E		

$$h(x) = 2$$

2	A	2	C
5	B	5	H

Partitions for S

$$h(x) = 0$$

9	w
6	t

$$h(x) = 1$$

1	z	1	x
4	r	4	u

$$h(x) = 2$$

2	v
5	s

S

1	z
4	r
9	w
1	x
2	v
4	u
6	t
5	s

$$M=4, bfr=2, h(x) = x \% 3$$

Hash Join Example

Join Phase on Partition 1



Partition 1 for R

$$h(x) = 1$$

1	D	4	G
7	E		

Partition 1 for S

$$h(x) = 1$$

1	z	1	x
4	r	4	u

Buffers

Blocks
of R

1	D
7	E
4	G

Last block
for S

1	z
4	r

Output

1	D	z
4	G	r

Put all of R into memory buffers (build relation).
Read 1 block of S at a time.

Hash Join Example

Join Phase on Partition 1 (block #2 of S)



Partition 1 for R

$$h(x) = 1$$

1	D	4	G
7	E		

Partition 1 for S

$$h(x) = 1$$

1	z	1	x
4	r	4	u

Buffers

Blocks
of R

1	D
7	E
4	G

Last block
for S

1	x
4	u

Output

1	D	x
4	G	r
1	D	x
4	G	u

Read second block of S. Compute join output.

Hash-Join Analysis

The hash-join algorithm performs $3*(B(R)+B(S))$ disk I/Os.

- $2*B(R)+2*B(S)$ to perform the hash partitioning on the relations.
- $1*B(R)+1*B(S)$ as each block of each relation read in to perform join (one bucket at a time).

Algorithm limited to relations where $\min(B(R), B(S)) \leq M^2$.

Hash-Join Example

Let relations R and S occupy 6,000 and 3,000 blocks respectively. Let $M = 101$ blocks.

Hash-join algorithm cost:

$$= 3 * (B(R) + B(S)) = \text{27,000 disk I/Os}$$

- Average # of blocks per bucket is 60 (for R) and 30 (for S).
- Algorithm works because $\min(60, 30) \leq 101$.

Hybrid-Hash Join Algorithm

Hybrid hash join uses any extra space beyond what is needed for buffers for each bucket to store one of the buckets in memory. This reduces the number of I/Os. Idea:

- Assume that we need k buckets in order to guarantee that the partitions of the smaller relation S fit in memory after partitioning.
- Of the M buffers, allocate $k-1$ buffers for each of the buckets except the first one. Expected bucket size is $B(S)/k$.
- Give bucket 0 the rest of the buffers ($M-k+1$) to store its tuples in memory. The rest of the buckets are flushed to disk files.
- When hash relation R , if tuple t of R hashes to bucket 0, we can join it immediately and produce output. Otherwise, we put it in the buffer for its partition (and flush this buffer as needed).
- After read R , process all on-disk buckets using a one-pass join.

Hybrid-Hash Join Analysis

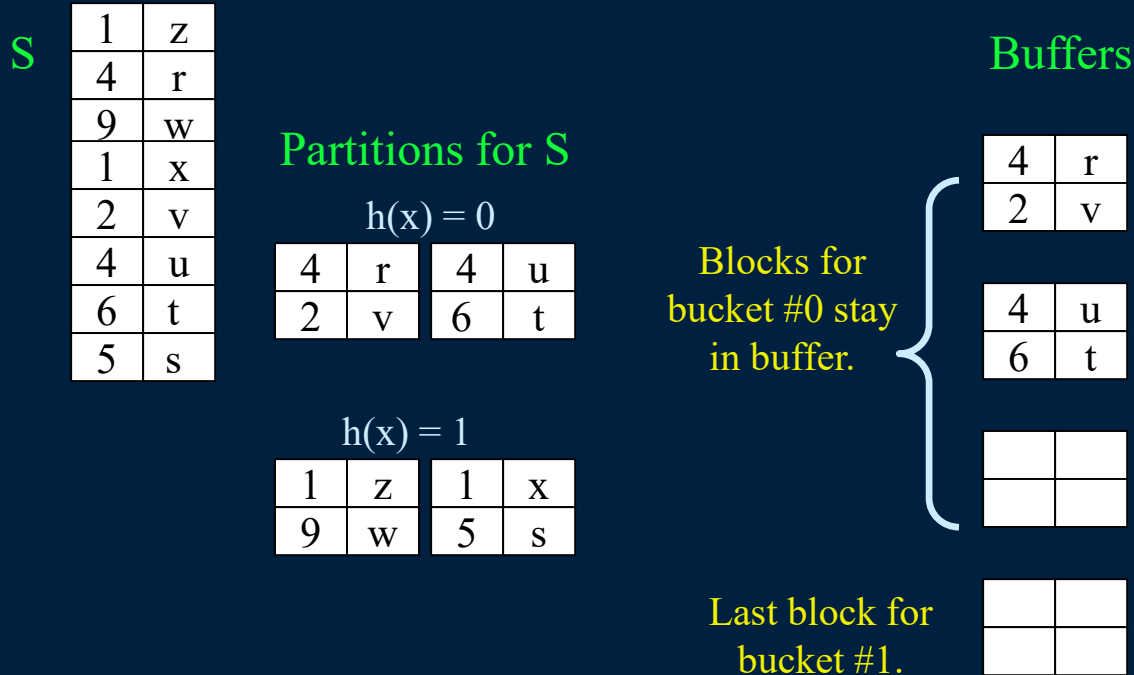
This approach saves two disk I/Os for every block of the buckets of S that remain in memory.

Overall cost is: $(3 - \frac{2M}{B(S)})(B(R) + B(S))$

- Note: We are making the simplification that the in-memory partition takes up all of memory M (in practice it gets $M-k+1$) buffers. This is usually a small difference for large M and small k .

Hash Join Example

Partition Phase



M=4, bfr=2, buckets=2
Keep bucket 0 in memory.
Bucket 0 can use up to 3 blocks.

Hash Join Example

Buffered Join Phase



R

2	A	0
5	B	1
2	C	0
1	D	1
7	E	
3	F	
4	G	
5	H	

Buffers

4	r
2	v

4	u
6	t

5	B
1	D

Output

2	A	v
2	C	v

Partition R.
Join immediately if hash
to bucket 0.

$$h(x) = 1$$

On Disk

Hash Join Example

Buffered Join Phase (2)



R

2	A
5	B
2	C
1	D
7	E
3	F
4	G
5	H

1
1

Partition R.
Join immediately if hash
to bucket 0.

$$h(x) = 1$$

Buffers

4	r
2	v

4	u
6	t

7	E
3	F

Output

2	A	v
2	C	v

On Disk

5	B
1	D

Hash Join Example

Buffered Join Phase (3)



R

2	A
5	B
2	C
1	D
7	E
3	F
4	G
5	H

0
1

Partition R.
Join immediately if hash
to bucket 0.

$h(x) = 1$

Buffers

4	r
2	v

4	u
6	t

5	H

Output

2	A	v
2	C	v
4	G	r
4	G	u

On Disk

5	B
1	D

7	E
3	F

Hash Join Example

Disk Join Phase



Perform regular hash join on partition 1 of R and S currently on disk.

Partition 1 On Disk for R

5	B	7	E	5	H
1	D	3	F		

Partition 1 On Disk for S

1	z	1	x
9	w	5	s

Buffers

1	z
9	w

1	x
5	s

5	H

Blocks of S.

Buffer 1 block of R at a time.

Output

2	A	v
2	C	v
4	G	r
4	G	u
5	B	s
1	D	z
1	D	x
5	H	s

Hash-Join Example Analysis

Hash-join algorithm cost **26 total block I/Os**. (Expected 24)

- Total partition cost = 17 I/Os.
 - Partition of R : 4 reads, 5 writes.
 - Partition of S : 4 reads, 4 writes.
- Join phase cost = 9 reads (5 for R and 4 for S).
- Total cost of 26 is larger than expected cost of 24 because tuples did not hash evenly into buckets.

Hybrid-hash join algorithm cost **16 block I/Os**. (Expected 16)

- Partition cost is 12 disk I/Os.
 - Partition of R : 4 reads, 2 writes (for bucket #1) (do not write last block).
 - Partition of S : 4 reads, 2 writes.
 - Memory join is free.
- Regular hash join: 2 read for R , 2 reads for S .

Hash Join Question

Question: Select a true statement.

- A)** The probe relation is the smallest relation.
- B)** The probe relation has an in-memory hash table built on its tuples.
- C)** The build relation is the smallest relation.
- D)** The probe relation is buffered in memory.

Multi-Pass Hash Joins

We have examined two-pass hash joins where only one partitioning step is needed. Hash-based joins can be extended to support larger relations by performing recursive partitioning.

Unlike sort-based joins where the number of partition steps is determined by the larger relation, for hash-based joins the number of partition steps is determined by the smaller build relation. This is often a significant advantage.

Adaptive Hash Join

During its execution, a join algorithm may be required to give up memory or be given memory from the execution system based on system load and execution factors.

An **adaptive hash join** algorithm [Zeller90] is able to adapt to changing memory conditions by allowing the partition buckets to change in size.

Basic idea (that makes it different from hybrid hash):

- Each partition can hold a certain number of buffers and all are initially memory resident. Tuples are inserted as usual.
- When memory is exhausted, a victim partition is flushed to disk and frozen (no new tuples can be added). This is repeated until partitioning is complete.

The description of adaptive join algorithm above is for the simpler version called dynamic hash join [DeWitt95].

Local Research

Skew-Aware Hash Join



Skew-aware hash join [Cutt09] selects the build partition tuples to buffer based on their frequency of occurrence in the probe relation.

When data is skewed (some data is much more common than others), this can have a significant improvement on the number of I/Os performed.

Algorithm optimization is currently in PostgreSQL hash join implementation.



Summary of Hashing Based Methods

Performance of hashing based methods:

Operators	Approximate M required	Disk I/Os
γ, δ	\sqrt{B}	$3 * B$
$\cup, -, \cap$	$\sqrt{B(S)}$	$3 * (B(R) + B(S))$
\bowtie (simple)	$\sqrt{B(S)}$	$3 * (B(R) + B(S))$
\bowtie (hybrid)	$\sqrt{B(S)}$	$(3 - \frac{2M}{B(S)})(B(R) + B(S))$

Comparison of Sorting versus Hashing Methods



Speed and memory requirements for the algorithms are almost identical. However, there are some differences:

- 1) Hash-based algorithms for binary operations require memory based on the size of the smaller of the two relations rather than the sum of the relation sizes.
- 2) Sort-based algorithms produce the result in sorted order which may be used for later operations.
- 3) Hash-based algorithms depend on the buckets being of equal size.
 - Hard to accomplish in practice, so generally, we limit bucket sizes to slightly smaller values to handle this variation.
- 4) Sort-based algorithms may write sorted sublists to consecutive disk blocks.
- 5) Both algorithms can save disk access time by writing/reading several blocks at once if memory is available.

Comparison of Sorting versus Hashing Methods (2)



- 6) Hash based joins are usually best if neither of the input relations are sorted or there are no indexes for equi-join.

Note that for small relation sizes, the simple nested-block join is faster than both the sorting and hashing based methods!

Sort/hash duality:

- Hashing performs divide and conquer.
- Sorting performs conquer and merge.

Join Question

Question: Assume the percentage of join memory available compared to the smaller relation size is $M / B(S)$. Select a percentage where block nested-loop join is faster than hybrid hash join in terms of disk I/Os.

- A) 10%
- B) 25%
- C) 40%
- D) 70%
- E) 100%

Index-Based Algorithms

Index-based algorithms use index structures to improve performance.

Indexes are especially useful for selections instead of performing a table scan.

For example, if the query is $\sigma_{a=v}(R)$, and we have a B+-tree index on attribute a then the query cost is the time to access the index plus the time to read all the records with value v .

Cost Estimate Example with Indices

Query: $\sigma_{Major = \text{"CS"}}(Student)$

- Evaluate query cost assuming:
 - $V(Student, Major)=4$ (4 different Major values: "BA", "BS", "CS", "ME")
 - $B(Student) = 500$, $T(Student) = 10,000$, blocking factor = 20
- Cost estimate for query using *Major* index:
 - Since $V(Student, Major)=4$, we expect that $10000/4 = 2,500$ tuples have "CS" as the value for the *Major* attribute.
 - If the index is a clustering index, $2,500/20 = 125$ block reads are required to read the *Student* tuples. (What would be the strategy?)
 - If the index is non-clustering, how many index blocks are read?
 - The height of the index depends on the # of unique entries which is 4. The B+-tree index would be of depth 1. We can assume that it would be in main memory, only the pointer blocks would have to be read. If a leaf node can store 200 pointers, then $2,000/200 = 13$ index blocks would have to be read.
 - How many block I/Os in total for a non-clustering index?
 - How does this compare to doing a sequential scan?

Index-Based Algorithms

Complicated Selections



Indexes can also be used to answer other types of selections:

- 1) A B-tree index allows efficient range query selections such as $\sigma_{a \leq v}(R)$ and $\sigma_{a \geq v}(R)$.
- 2) Complex selections can be implemented by an index-scan followed by another selection on the tuples returned.

Complex selections involve more than one condition connected by boolean operators.

- For example, $\sigma_{a=v \text{ AND } b \geq 10}(R)$ is a complex selection.

This query can be evaluated by using the index to find all tuples where $a=v$, then apply the second condition $b \geq 10$ to all the tuples returned from the index scan.

Index Joins

An index can also be used to speed-up certain types of joins.

Consider joining $R(X,Y)$ and $S(Y,Z)$ by using a nested-block join with S as the outer relation and R as the inner relation. We have an index on R for attribute(s) Y .

We can modify the algorithm that for every tuple t of S , we use the value of Y for t to lookup in the index for R .

This lookup will return only the tuples of R with matching values for Y , and we can compute the join with t .

Cost: $T(S) * (T(R)/V(R,Y))$ tuples will be read

- $T(S) * T(R)/V(R,Y)$ (non-clustered)
- $T(S) * B(R)/V(R,Y)$ (clustered)
- Not always faster than a nested-block join! Makes sense when $V(R,Y)$ is large and S is small.

Index-Merge Join Variant Example

Join $R(X,Y)$ with $S(Y,Z)$ by sorting R and read S using index.

- with $B(R)=6000, B(S)=3000, M = 101$ blocks.

1) Assume only index on S for Y :

Sort R first = $2 * B(R) = 12,000$ disk I/Os (to form sorted sublists)

Merge with S using 60 buffers for R and 1 for index block for S .

Read all of R and $S = 9,000$ disk I/Os

Total = **21,000 disk I/Os**

2) Assume index for both R and S for Y :

Do not need to sort either R or S .

Read all of R and $S =$ **9,000 disk I/Os**

Remember that there is always a small overhead of accessing the index itself.

Multi-Pass Algorithms

The two-pass algorithms based on sorting and hashing can be extended to any number of passes using recursion.

- Each pass partitions the relations into smaller pieces.
- Eventually, the partitions will entirely fit in memory (base case).

Analysis of k -pass algorithm:

- Memory requirements $M = (B(R))^{1/k}$
- Maximum relation size $B(R) \leq M^k$
- Disk operations = $2 * k * B(R)$
 - Note: If do not count write in final k pass, cost is: $2 * k * B(R) - B(R)$.

Parallel Operators

We have discussed implementing selection, project, join, duplicate elimination, and aggregation on a single processor.

Many algorithms have been developed to exploit parallelism in the form of additional CPUs, memory, and storage.

Current Systems – Beyond Just I/Os

Current systems optimized for CPU and cache usage in addition to I/O as highest performance databases will be mostly memory resident.

Algorithms access cache line sized memory chunks in order to increase cache hit rate and reduce memory wait times.

Parallelization using vectorization in CPU and GPU increases performance.

Join Algorithms that Produce Results Early



One of the problems of join algorithms is that they must read either one (hash-based) or both (sort-based) relations before any join input can be produced.

- This is not desirable in interactive settings where the goal is to get answers to the user as soon as possible.

Research has been performed to define algorithms that can produce results early and are capable of joining sources over the Internet. These algorithms also handle network issues.

- **Sort-based algorithms:** Progressive-Merge join [Dittrich02] produces results early by sorting and joining both inputs simultaneously in memory.
- **Hash-based algorithms:** Hash-merge join [Mokbel04], X-Join [Urban00] and Early Hash Join [Lawrence05] use two hash tables. As tuples arrive they are inserted in their table and probe the other.

Research Challenges

There are several active research challenges for database algorithms:

- 1) Optimizing algorithms for cache performance
- 2) Examination of CPU costs as well as I/O costs
- 3) The migration to solid-state drives changes many of the algorithm assumptions.
 - Random I/O does NOT cost more any more which implies algorithms that performed more random I/O (index algorithms) may be more competitive on the new storage technology.

Conclusion

Every relational algebra operator may be implemented using many different algorithms. The performance of the algorithms depend on the data, the database structure, and indexes.

Classify algorithms by:

- 1) **# of passes:** Algorithms only have a fixed buffered memory area to use, and may require one, two, or more passes depending on input size.
- 2) **Type of operator:** selection, projection, grouping, join.
- 3) Algorithms can be based on sorting, hashing, or indexing.

The actual algorithm is chosen by the query optimizer based on its query plan and database statistics.

Major Objectives

The "One Things":

- Diagram the query processor components and explain their function (slide #5).
- Calculate block access for one-pass algorithms.
- Calculate block accesses for tuple & block nested joins.
- Perform two-pass sorting methods including all operators, sort-join and sort-merge-join and calculate performance.
- Perform two-pass hashing methods including all operators, hash-join and hybrid hash-join and calculate performance.

Major Theme:

- The query processor can select from many different algorithms to execute each relational algebra operator. The algorithm selected depends on database characteristics.

Objectives

- Explain the goal of query processing.
- Review: List the relational and set operators.
- Diagram and explain query processor components.
- Explain how index and table scans work and calculate the block operations performed.
- Write an iterator in Java for a relational operator.
- List the tuple-at-a-time relational operators.
- Illustrate how one-pass algorithms for selection, project, grouping, duplicate elimination, and binary operators work and be able to calculate performance and memory requirements.
- Calculate performance of tuple-based and block-based nested loop joins given relation sizes (memorize formulas!).

Objectives (2)

- Perform and calculate performance of two-pass sorting based algorithms - sort-merge algorithm, set operators, sort-merge-join/sort-join.
- Perform and calculate performance of two-pass hashing based algorithms - hash partitioning, operation implementation and performance, hash join, hybrid-hash join.
- Compare/contrast sorting versus hashing methods
- Calculate performance of index-based algorithms - cost estimate, complicated selections, index joins
- Explain how two-pass algorithms are extended to multi-pass algorithms.
- List some recent join algorithms: adaptive, hash-merge, XJoin, progressive-merge.



THE UNIVERSITY OF BRITISH COLUMBIA

