ER Diagram

- Many to Many

A can have multiple B and B can have multiple A

- One to Many

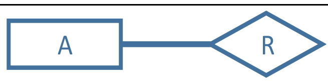
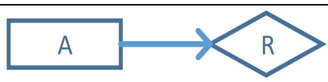
A can have multiple B but B can only have one A

- One to One

A can have only one B and B can have only one B

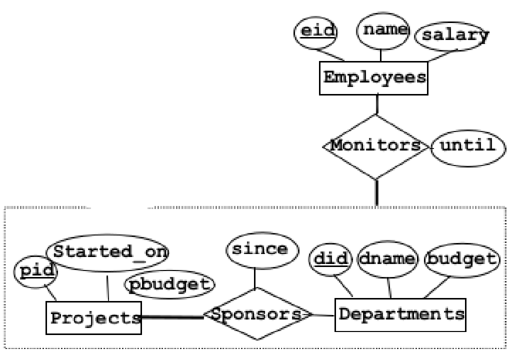
- At least one

Bold arrow specify there is at least one element.

	A have at least one
	A have exactly one

- Aggregation

Allows us to treat a reltionship set R as an entity set so that R can participate in other relationships



Relational algebra

Selection	$\sigma_{condition}(Element)$
Projection	$\pi_{attributes}(Element)$
Renaming	$\rho(R(A_1, .., A_n), R_{alias}(B_1, .., B_n))$
Cross product	$\times$
Join	$\bowtie$
Division	$\backslash$
Intersection	$\cap$
Union	$\cup$
Set different	$-$

- Condition/Theta Join $Rout = Rin1 \bowtie cRin2 = \sigma c(Rin1 \times Rin2)$
- Equi Join: $Rout = Rin1 \bowtie Rin1a1 = Rin2b1, \ldots, Rin1an = Rin2bnR$in2 Condition join where condition contains ONLY equalities
- Natural Join: Equijoin on all common attribute

Sql

Datatype

Char(n)	A character string of fixed length n

VarChar(n)	Denotes a string of up to n charaters
INT or INTEGER	An integer
SHORTINT	Smaller integer
FLOAT or REAL	Float number
DOUBLE PRECISION	Double
DATE	Date format YYYY-MM-DD
TIME	Time format: hh:mm:ss

Table operations

```
--Create table
CREATE TABLE Students
(
        id INT NOT NULL,
        name VARCHAR(20),
        login CHAR(10),
        major VARCHAR(20) DEFAULT 'undefined',
        school_id INT,
        PRIMARY KEY(id),
        FOREIGN KEY(school_id) REFERENCES School(id)
)

--Drop table
DROP TABLE Students

--Alter table
ALTER TABLE Students ADD COLUMN firstyear:integer
```

Row operation

```
--INSERT
INSERT INTO Students (id, name, faculty) VALUES (8908998, 'Dupont', 'Science')

--Delete
DELETE FROM Students WHERE id = 0894984

--Update
UPDATE Students SET faculty = 'Arts' WHERE id = 9849849
```

Trigger

```
CREATE TRIGGER updateSkater
 AFTER DELETE ON Skaters
REFERENCING OLD TABLE AS DeletedSkaters
FOR EACH STATEMENT
 INSERT
 INTO StatisticsTable(ModTable, ModType, Count)
 SELECT ‘Skaters’, ‘delete’, COUNT(*)
 FROM DeletedSkaters
```

```
Use begin/end to encapsulate more than one
action
FOR EACH ROW/STATEMENT
WHEN …
BEGIN ATOMIC
 do 1thing;
 do 2nd thing;
END
```

XML

WHAT DAFUQ?!!@#!@#!?

DTD

```
<!DOCTYPE DiscoverTheWorld [
<!ELEMENT DiscoverTheWorld (tour*,reservation*)>
<!ELEMENT tour (type, start-date, duration, price) >
<!ELEMENT reservation (cname, caddress, cost,
special*)>
<!ATTLIST tour TourId ID #REQUIRED >
<!ATTLIST reservation ResID ID #REQUIRED
TourID IDREF #REQUIRED>
<!ELEMENT type (#PCDATA) >
<!ELEMENT start-date (#PCDATA) >
<!ELEMENT duration (#PCDATA) >
<!ELEMENT price (#PCDATA) >
<!ELEMENT cname (#PCDATA) >
<!ELEMENT caddress (#PCDATA) >
<!ELEMENT cost (#PCDATA) >
```

```
<!DOCTYPE Politics [ <!ELEMENT Politics (Politician*,
Province*)>
<!ELEMENT Politician ((CurrentMayor | CurrentMop)?,
address?)>
<!ELEMENT CurrentMayor (since?)>
<!ELEMENT CurrentMoP (since?)>
<!ELEMENT Province (City+, Riding+, population?)>
<!ELEMENT City (population?)>
<!ELEMENT Riding (population?)>
<!ELEMENT address (#PCDATA) >
<!ELEMENT since (#PCDATA) >
<!ELEMENT population (#PCDATA) >
<!ATTLIST Politician pname ID REQUIRED
website CDATA IMPLIED
friends IDREFS IMPLIED>
```

```
<!ELEMENT special (#PCDATA) >
<!ATTLIST special price CDATA #REQUIRED>
]>
```

```
<!ATTLIST CurrentMayor cityID IDREF REQUIRED>
<!ATTLIST CurrentMoP rname IDREF REQUIRED>
<!ATTLIST Province pname ID REQUIRED>
<!ATTLIST City cityID ID REQUIRED
cname CDATA REQUIRED>
<!ATTLIST Riding rname CDATA REQUIRED>
]>
```

```
<bibliography>
        <books>
                <book ISBN="23456" year="1995">
                        <title> Foundations ...
</title>
                        <author> Hull </author>
                        <author> Abiteboul </author>
                        <publ> Addison Wesley </publ>
                </book>
                <book> ...</book>
        </books>
        <journals>
                <journal>
                        <title> ... </title>
                        <article> ... </article>
                        ...
                </journal>
                <journal> ... </journal>
        </journal>
</bibliography>
```

```
<DiscoverTheWorld>
        <tour TourId="1">
                <type> Brazil junge </type>
                <start-date> 16-April </start-date>
                <duration> 14 </duration>
                <price> 2229 </price>
        </tour>
        <tour TourId="2">
                <type> Brazil junge </type>
                <start-date> 30-April </start-date>
                <duration> 21 </duration>
                <price> 2999 </price>
        </tour>
        <tour TourId="3">
                <type> Kenia safari </type>
                <start-date> 30-April </start-date>
                <duration> 21 </duration>
                <price> 3229 </price>
        </tour>
        <reservation ResId="541" TourId="1">
                <cname> Bettina Kemme </cname>
                <caddress> Montreal </caddress>
                <cost> 2579 </cost>
                <special price="5"> vegetarian </special>
                <special price="20"> single </special>
        </reservation>
        <reservation ResId="542" TourId="2">
                <cname> Your Name </cname>
                <caddress> Your Address </caddress>
                <cost> 3105 </cost>
                <special price="5"> vegetarian </special>
        </reservation>
</DiscoverTheWorld>
```

```
<!DOCTYPE people[
    <!ELEMENT people(person*)>
    <!ELEEMNT person(name*, (lastname|familyname)?)>
    <!ATTLIST person PID ID #REQUIRED
        age CDATA #IMPLIED
        children IDREFS #IMPLIED
        mother IDREF #IMPLIED
    >
    <!ELEMENT name(#PCDATA)>
    <!ELEMENT lastname(#PCDATA)>
    <!ELEMENT familyname (#PCDATA)>
]>
```

Data types: PCDATA (parsed character data) or CDATA (unparsed)

Attributes

- ID unique identifier (similar to primary key)
- IDREF: reference to single ID
- IDREFS: space-seperated list of references

Values

- can give a default value
- #REQUIRED must exist
- #IMPLIED optional

Specified in an XML file with <!DOCTYPE name SYSTEM "path/to/thing.dtd">

Can use regex style things too. * is 0 or more. + is 1 or more, (a | b)? is one or the other

XPATH

- /bibliography/book/author all author elements by root navigating through those elements
- /bibliography/book/@ISBN All ISBN attributes
- //title all title elements anywhere in the document
- /bibliography/*/title titles of bibliography entries assuming that there could be books, journals, reports, etc...
- /bibliography/book[@year>1995] returns books where the year > 1995
- /bibliography/book[author='FooBar']/@Year returns the years of books written by FooBar
- /bibliography/book[count(author) <2]
- /bibliography/book/author[position()=1]/name position is the location of the node in the node set

XQuery

For	Let
for $b in document("bib.xml")/bib/book return <result> $b</result>	let $b in document("bib.xml")/bib/book return <result> $b</result>
<result><book>…</book></result> <result><book>…</book></result> <result><book>…</book></result> … <result><book>…</book></result>	<result> <book>…</book> <book>…</book> . . <book>…</book> </result>

-Basic Queries: **'/'** to navigate one path at a time
Example:/Bookstore/Book/Title
'//' all paths following this
Example://Title
When wanting to access an attribute of an element use @
Example:/Bookstore/Book/data(@ISBN)
'|' OR operator ONLY USED INSIDE CONDITIONS
Example:/Bookstore/Book|Magazine/Title
'=' can act like == like in Self-Join Queries below
Navigation accesses:
Example: parent::* *::child following-silbling::*

-Queries involving CONDITIONS

1condition:
Example:/Bookstore/Book[@Price<30]
Example:/Bookstore/Book/Authors/Author[2]
the 2nd author of each element

2conditions:
To write a condition, it needs to be inside '[...]' then followed by the output that we are looking for
Example:/Bookstore/Book[@Price<30]/Title

Condition to find elements that contain other elements:
Example:/Bookstore/Book[Remark]/Title

Conditions && Conditions + Some output:
Example: Looking for a title that has one last name =Ullman and price<90
/Bookstore/Book[@Price<90 and Authors/Author/Last_Name="Ullman"]/Title

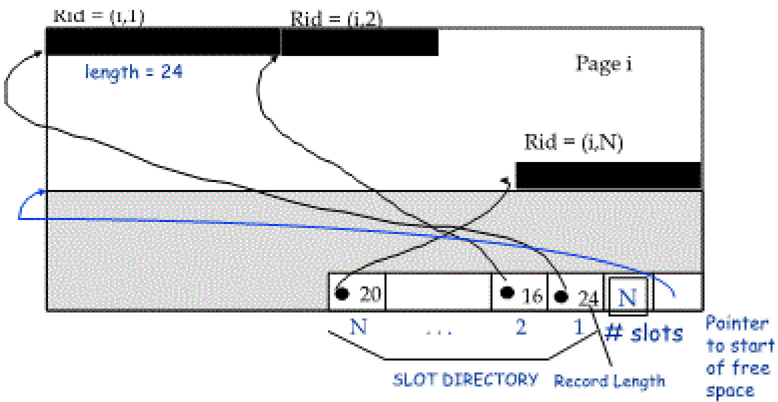
Conditions Inside Conditions + Some output:
Example:Looking for a title with author ="Jeffrey Ullman" and price<90
/Bookstore/Book[@Price<90 and Authors/Author/[Last_Name="Ullman" and First_Name="Jeffrey"]]/Title

Conditions && !Conditions + Some output:
Example:Looking for a title with author ="Ullman" and NOT author="Widom"
/Bookstore/Book[/Authors/Author/Last_Name="Ullman" and count(/Authors/Last_Name="Widom"=0]/Title

The condtion 'contains':
/Bookstore/Book[contains(Remark, "great")]/Title

Self-Join Query
Quering two instances of the database at one and joining them together.Trying to find the magazines wheres theres a book with the same title. Example:
doc("BookstoreQ.xml")/Bookstore/Magazine[Title=doc("BookstoreQ.xml")/Bookstore/Book/Title]

Navigation Accesses
The name() function returns the name of a tag or element
To find all elements whose parent is not "Bookstore" or "Book"
/Bookstore/*[name(parent::*)!="Bookstore" and name(parent::*)!="Book"]



- **Record id (rid)** = internal identifier of a record: <page id, slot #>.
- Can move records on page without changing rid;

XML in DB@^%$^$#

```
INSERT INTO MyXML(id, INFO) VALUES (1000,
    '<customerinfo cid="1000">
    <name>Kathy Jones</name>
    <addr country =Canada">
        <street>123 fake</street>
        <city>Ottawa</city>
        <prov-state>Ontario</prov-state>
        <pcode-zip>H0H 0H0</pcode-zip>
    </addr>
    </customerinfo>')
```

Buffer

DBMS stores information persistently on (“hard”) disks. ❑ Unit of transfer main-memory/disk: disk blocks or pages. ❑ Timing: ☆ 2- 20 msec for random data block (bad seek time) ☆ If blocks are sequentially on disk, only +1ms per block ☆ Compare main memory access: in nanoseconds ❑ Basic operations (READ/WRITE from/to disk) ❑ Why disks? ☆ Cheaper than Main Memory ☆ Higher Capacity ☆ Main Memory is volatile	When loading a page from disk: ☆ Replacement frame must have “pin counter” of 0 ❑ When requesting a page that is in the buffer ☆ Increment pin counter ❑ After operation has finished ☆ Decrement pin counter ☆ Set dirty bit if page has been modified: ❑ Frame is chosen for replacement by a replacement policy: ☆ Only unpinned page can be chosen (pin count = 0) ☆ Least-recently-used (LRU), Clock, MRU etc.	If requested is not in pool: ☆ If there is an empty frame, use it ☆ Else choose an empty frame for replacement. If the frame is dirty (page was modified), write it to disk ☆ Read requested page into chosen frame Buffer management in DBMS requires ability to: ☆ **pin a page** in buffer pool, **force a page to disk** (important for implementing CC & recovery), ☆ adjust **replacement policy**, and **pre-fetch pages** based on access patterns in typical DB operations.

Indexing

COST MODEL	HEAP FILES	SORTED FILES
Measure performances by simplifying the parameters (IO focused): ☆ only consider disk reads (ignore writes) ☆ only consider number of I/Os and not the individual time for each	☆ Linked, unordered list of all pages of the file ☆ Is it good for: ● scan retrieving all records (SELECT *)? ▲ yes, you have to retrieve all	☆ Records are ordered according to one or more attributes of the relation ☆ Is it good for: ● scan retrieving all records (SELECT *)?

read (ignores page pre-fetch) ☆ Average-case analysis; based on several simplistic assumptions. ● delete/update ▲ depends on where	pages anyway ● equality search on primary key ▲ not great: have to read on avg half the pages for 1 record ● range search or equality search on non-primary key ▲ not great, all pages need to be read ● insert ▲ yes, can insert anywhere ● delete/update ▲ depends on where	▲ yes, you have to retrieve all pages anyway ● equality search on sort attribute ▲ good: find first qualifying page with binary search (log2) ● range search on sort attribute ▲ good: find first qualifying page with binary search (log2): adjacent pages might have additional matching records

Let suppose we have a relation R (A, B, C, D, F) such that:

- A and B are int (6 byte)
- C-F are char [40] (10 byte per char).
- Tuple = 172 bytes. 200,000 tuples
- Each data page has 4000 bytes and is around 80% full
- B values are uniformly distributed
- Rid = 10 bytes
- Size of pointer in intermediate page = 8 bytes
- Index pages are 4K and between 50%-100% full

Goal	Formula	With this example
Number of pages	$\frac{number\ of\ tuples\ *\ tuple\ size}{fill\ rate\ *\ page\ size}$	$\frac{172*200000}{40000*0.80} = 10750$
Index entry size in root and intermediate pages	$size\ of\ key + size\ of\ pointer$	$6 + 8 = 14\ bytes$
Average number of rids per data entry	$\frac{\text{number of tuples}}{\text{different values (if uniform)}}$	$\frac{200{,}000}{20{,}000} = 10$
Average length per data entry	$\text{size of key} + (\text{number of rids} * \text{size of rid})$	$6 + 10*10 = 106$
Average number of data entries per leaf page	$\frac{\text{fillrate} * \text{page size}}{\text{length of data entry}}$	$\frac{0.75*4000}{106} = 28\ entries\ per\ page$
Estimate number of leaf page	$\frac{\text{number of different values}}{\text{number of entrier per page}}$	$\frac{20{,}000}{28} = 715$
Number of entries in intermediate pages	$\frac{\text{fillrate} * \text{page size}}{\text{lenght of index enty}}$	$min = \frac{0.5*4000}{14} = 143, \max = \frac{1*4000}{14} = 285$
Height of tree	$(nb\ of\ entry\ in\ intermediate\ page)^{h-1} > nb\ of\ leaf\ page$	3

Non-clustered index B-tree with <k, list of rid>

Height of tree = Number of leaf pages / (min | max)? number of entries in intermediate pages

Give the pids of all projects within department D2 that started in 2014.

$$\pi_{pid}\left(\sigma_{dep_{id}=D2 \wedge start_{date}=2014}(Project)\right)$$

Give the pids of all projects that have at least one excellent evaluation

$$\pi_{Project.pid}\left(\sigma_{Evaluation.grade='execlent'}(Project \bowtie Evaluation)\right)$$

Force Flush strategy • All changes are flush to disk BEFORE commit	• Completed transaction need not action • Active transaction might have partial changes on disk(Need undone)	• Append to log file log record before flushing • At commit/abort append to log file commit/abort log record • When recovering from crash: Scan log backward for each record if commited ignore otherwise install Before-Image of the record
No force flush strategy • Changes might be flushed at any time(BEFORE/AFTER commit)	• Done transaction might have missing changes (must be redone) • Active/Aborted transaction might have been flushed before crash(Must be undone)	• For each write(x) of a transaction T with x being on page P: Log record with before AND after image of x(Before so you can undo changes, After so you can redo changes) • Flush before-image to disk before flushing the P • Flush after-image to disk before commit of T • At commit/abort append commit/abort record to log file and flush

• **Unrepeatable read:** If T1 read twice the same data item but T2 change its value between the first and the second • **Dirty read:** If T2 read from T1 before T1 commit. • **Lost update:** If T2 modify a data item modified by T1 without taking in account the value modified by T1.	

Schedule • Serial schedule: All transaction one after the other • Non-serial schedule: Transaction overlap - Serializable: Dependency graph has no cycle(T1 always does action before T2) - Recoverable schedule: If transaction T_i reads a value written by transaction T_j then T_i commit only after T_j committed - Avoiding cascading aborts: A transaction reads only values written by committed transactions. - Strict: A transaction only read or overwrite value written by committed transaction	Schedule examples: • Strict and serializable $r1(x), w2(x), c2, w1(x), c1$ • Avoids cascading aborts, non-strict, serializable • Recoverable, not avoiding cascade aborts, serializable $r1(x), w2(y), w2(x), r1(y), c2, c1$ • Not recoverable, serializable $r1(x), w2(y), r1(x), c1, c2$ • Not recoverable, Not-serializable

Unrecoverable		**Recoverable schedule with cascading abort**		**Recoverable schedule with commit**		**Avoids cascading**		**Non strict**		**Strict**		**Strict and serializable**	
T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2	T1	T2
R(A)		R(A)		R(A)		R(A)		W(A)		W(A)		R(x)	
W(A)		W(A)		W(A)		W(A)			W(A)	abort			W(x)
	R(A)		R(A)		R(A)	abort		abort			W(A)		commit
	commit	abort		commit			R(A)		commit		commit	W(x)	
commit			abort		commit		commit					commit	

Lock request: • If lock is S, no X lock is active and the request queue is empty: - Add the lock to the granted lock queue and set the lock type to S • If lock is X and no lock active(request queue is also empty): - Add the lock to the granted lock queue and set the lock type to X • Otherwise - Add the lock to the request lock queue	**Lock release:** • Remove the lock from the granted lock queue • If this was the only lock granted on this object: - Grant one X lock(If the first of the request is a X lock) - Grant n S lock(If the first n element are S lock)
Deadlocks: • Make the wait-for graph(T_i $need\ ressource\ lock\ by\ T_j$) • If cycle then we have a deadlock (Nooooooooooooo…)	**Solve Deadlock:** • Add a timeout for each transaction and abort if transaction timeout. Problem on what timeout value to choose • Request all the lock at the beginning of the transcation

Predicate locking:
• Grant lock on all records that satisfies logical predicates(e.g. depid>5, age > 2*salary)
• More bullshit
	Predicate looking example:
	• Assume 2 tranascrtions:
	- $UPDATE\ Skaters\ set\ rating\ =\ 7\ WHERE\ sid\ =\ 123$
	- $SELECT\ max(age)\ FROM\ Skaters\ WHERE\ rating\ =\ 5$
	• Assume: T1 execute first then it has a X-lock on Skaters with sid=123
	• Assume: T2 has to scan the entire table to get skater with rating=5
	- For each tuple
	  - set S-lock on tuple
	  - Check condition
	  - If condition TRUE keep lock and return value
	  - If condition FALSE release lock
	- It need to read the tuple where sid=123 and rating= 5 but block has T1 has a lock on it.
	- T2 is block by T1 although there is no conflict

Problems of strict 2PL locking:

- Very restrictive, low concurrency, problem with long query
- More and more exception

In order to allow for more concurrency, SQL2 defines various levels of isolation

- Assumed to be implemented by different forms of locking
- Avoid different levels of anomalies
- Used for non-critical transactions or read-only transactions
- Lower levels of isolation do NOT provide serializability

Problem

- Definitions are no more appropriate if systems do not use locking but other forms of concurrency control
- For instance, Oracle's "serializable" level does not provide serializable schedule as defined in the literature

Isolation level:

- In principle isolation levels are independent of concurrency control mechanics
- In reality they were defined with locking in mind

Isolation level/Anomaly	Dirty read	Unrepeatable read	Phantom
Read uncommitted	Maybe	Maybe	Maybe
Read committed	No	Maybe	Maybe
Repeated reads	No	No	Maybe
Serializable	No	No	No

- **Read uncommitted:**
  - Read op. do not set locks; can read not-committed updates
- **Read committed:**
  - Read op. set short S locks; have to wait for X locks to be released
  - release lock immediately after execution of op
- **Repeated reads:**
  - Read operations set standard lock S locks; standard 2PL
- **Serializable:**
  - Read op. must set S locks that cover all objects that are read
  - predicate locks or coarse locks (e.g., lock on entire relation)

Big data

Some bullshit info:

- Hardware
  - CPU does not increase
  - Instead muticode
- Usage
  - Astronomy: high-resolution, high-frequency sky surveys
  - Medicine: digital records, MRI, ultrasound
  - Biology: sequencing data
  - User behavior data: click streams, search logs

Parallel Query Evaluation:

- Inter-query parallelism
  - Different queries run in parallel on different processors; each query is executed sequentially
- Inter-operator parallelism
  - Different operator within the same execution tree run on different processors
- Intra-operator parallelism
  - A single operator(JOIN, GROUP, …) runs on many processor

Horizontal data Partitioning:

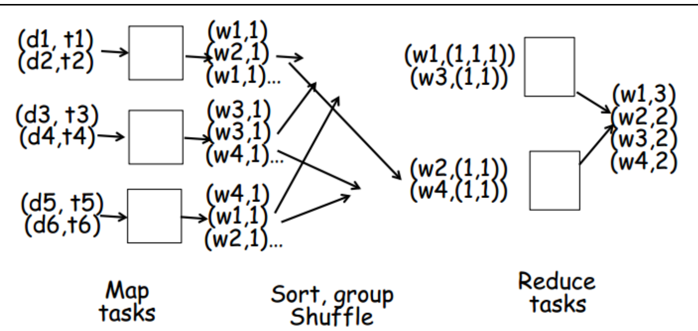
- Data
  - Large table $R(K, A, B, C)$
  - Key value store $KV(K, V)$
- Goal
  - Partition into chunks $c_1, \ldots c_n$ of records stored at N nodes
- Range partition
  - Equal size of each chunk
- Hash partitioned on attribute X
  - Record r goes to chunk i, according to hash function
  - xample: hash function H(r.X) mod P+1
- Range partitioned on attribute X
  - Partition range of X into: $-\infty = v_0 < v_1 < \cdots < v_p = \infty$
  - Record r goes to chunk i, if $v_{i-1} < t.X < v_i$

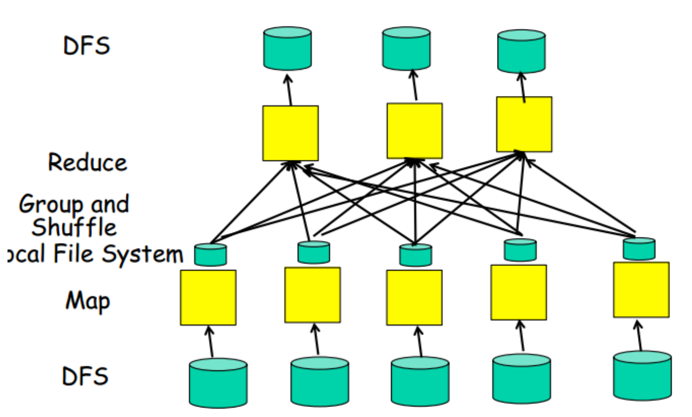
Vertical Partitioning:

- Column stores
- Data: relation $R(K, A, B, C)$
- Partition into $RA(K, A)$, $RB(K, B)$, $RC(K, C)$
- Query:
  - SELECT A FROM R
- Query only needs to access partition RA
- Much less IO

Execution steps:

- User indicates m (number of map tasks), r (number of reduce tasks), key/value set = document Set DS
- System creates m map tasks and splits input set of 1key/value pairs into m partitions and gives each map task one partition as input
- Each Map task executes user written map function
  - WordCountMap:
  - For each input key/value pair (dkey, dtext)
    - For each word w of dtext
      - Output key-value pair (w, 1)
- Next step only completes once all map tasks have completed
- System sorts map outputs by key and transforms all key/value pairs $(k, v1), (k, v2), \ldots (k, vn)$ with same key k to one key/value-list pair $(k, (v1, v2, \ldots vn))$
  - For Word count: all ('and', 1), ('and', 1), ('and', 1) … are transformed into one ('and', (1,1,1,….))
- System partitions output by key into r partitions and assigns these partitions as inputs to the r reduce tasks
- Each reduce task executes user written reduce function
  - WordCountReduce:
  - For each input key/value-list pair $(k, (v1, v2, \ldots\ vn)$
    - Output (k, n)





Map reduce

Relational Operators with Map/ reduce

- Assume R(A, B, C) relation (no duplicates)
- **Selection with condition c on R**
  - for each tuple t of R for which condition c holds, output (t, t)

- **Grouping: SELECT a, sum(b) GROUP by (a)**
  - Map: for each tuple (a, b,c) of R output (a,b)
  - Group and shuffle will create for each value a a key/value-list $(a, (b1, b2, \ldots)$

- Reduce: identity, that is output (t, t) • **Projection on A, B, of R** - Map: transform each tuple $t = (a, b, c)$ of R into tuple $t' = (a, b)$ of R, and output (t', t') - There might now be duplicates, that is several (t', t') tuples, the group function will aggregate them to $(t', (t', \ldots, t'))$ - Reduce: for each tuple $(t', (t', \ldots, t'))$ output (t', t')	- Reduce: for each $(a, (b1, b2, \ldots))$ perform aggregation $(e.g., b1 + b2, \ldots)$
• **Natural Join R(A,B,C) with Q(C,D,E) via hash-join(SELECT FROM R1,R2)** - Map: For each tuple (a, b, c) of R, output $(c, (R, (a, b)))$ For each tuple (c, d, e) of Q, output $(c, (Q, (d, e)))$ - Group and shuffle will aggregate all key/value pairs with same c-value - Reduce: For each tuple (c, value-list), example: $(value - list\ =\ (R, (a1, b1)), (R, (a2, b2)), \ldots (Q, (d1, e1)), \ldots))$ Rt = Qt = empty; for each v=(rel,tuple) in value-list if v.rel = R: insert tuple into Rt else insert tuple into Qt for v1 in Rt, for v2 in Qt, output(c,v1,v2) Basically produces all combinations (c, ai,bi,dj,ej)	Pig latin: • Users = **LOAD** 'users' **AS** (name,age); • Filtered = **FILTER** Users **BY** age >= 18 **AND** age <= 25; • Pages = **LOAD** 'pages' **AS** (uname, url); • Joined = **JOIN** Fltrd **BY** name, Pages **BY** uname; • Grouped = **GROUP** Jnd **BY** url; • Smmd = **FOREACH** Grpd **GENERATE** ($0), COUNT($1) **AS** clicks; • Srtd = **ORDER** Smmd **BY** clicks desc; • Top5 = **LIMIT** Srtd 5; • **STORE** Top5 **INTO** 'top5sites'

Query evaluation

Examples for flowing problems:

- Participates
  - 100 000 tuples
  - 1000 pages
  - 100 tuples per page
- Skaters
  - 40 000 tuples
  - 500 pages
  - 80 tuples per page
  - Index on sid has 170 leaf page
  - Index on names has 300 leaf page

- **Reduction factor of a condition defined as**
  - $Red(\sigma_{condition}(R)) = |\sigma_{condition}(R)|/|R|$
  - $Red\left(\sigma_{rating=5}(Skaters)\right) = \frac{|\sigma_{rating=5}(Skaters)|}{|Skaters|} = \frac{15}{100} = 0.15$
- **If not known, DBMS makes simple assumptions**
  - $Red\left(\sigma_{rating=5}(Skaters)\right) = \frac{1}{|diff\ rating|} = 0.1$
  - $Red\left(\sigma_{age\leq 10}(Skaters)\right) = \frac{10-\min(age)}{\max(age)-\min(age)} = \frac{10-4}{30-4} = \frac{6}{26} = 0.025$
- **Result size:** $number\ of\ input\ tuples * reduction\ factor$
- How to know the number of different values, max, min,
  - through indices, heuristics, separate statistics (histograms)

Simple selection:

- No index:
  - Search on arbitrary attributes: scan the entire relation
    e.g. $cost = page(Skaters)$
  - Search on primary key attributes: scan on average half of S
    e.g. $cost = \frac{page(Skaters)}{2} = 250$
- Index on selection attribute
  - Use index to find qualifying data entries, then retrieve corresponding data records.

Clustered B+tree

- Costs:
  - $\#LeafPages + \#dataPageToRetreive$
- Example 1
  - SELECT * FROM Skaters WHERE name sid = 5
  - 1 tuple match
  - $cost = 1\ leaf\ page + 1\ data\ page = 2$
- Example 2
  - SELECT * FROM Skaters WHERE name LIKE 'Z%'
  - System estimate the number of matching tuples(Around 100 match on 2 data page as its clustered)
  - $cost = 1\ leaf\ page + 2\ data\ page = 3$
- Example 3
  - SELECT * FROM Skaters WHERE name < 'F%'
  - Assume around 10 000 tuples match(On 125 datapage)
  - $cost = 1\ leaf\ page + 125\ data\ pages = 126$

Unclustered B+tree

- Costs:
  - $\#dataPageToRetreive = \#tuples$
  - $\#LeafPages + \#dataPageToRetreive$
- Example 1
  - SELECT * FROM Skaters WHERE name sid = 5
  - Same
- Example 2
  - SELECT * FROM Skaters WHERE name LIKE 'Z%'
  - Assume around 100 match on 80 data page but we need to retrieve some data page twice
  - $cost = 1\ leaf\ page + 101\ data\ page = 101$
- Example 3
  - SELECT * FROM Skaters WHERE name < 'F%'
  - Assume around 10 000 tuples match
  - $cost = 75\ leaf\ page + 10\,000\ data\ pages = 10\,001$

Unclustered B+tree with sorting:

- Sort matching data entries (rid=pid,slot-id) in leaf-pages by page-id
- Only fast if the the 75 leaf pages with matching entries fit in main memory
- Retrieve each page only once and get all matching tuples
- #data pages = #data pages that have at least one matching tuple;
- worst case is total # of data pages
- Example 1
  - SELECT * FROM Skaters WHERE name sid = 5
  - Same
- Example 2
  - SELECT * FROM Skaters WHERE name LIKE 'Z%'
  - Assume around 100 match on 80 data page
  - $cost = 1\ leaf\ page + 80\ data\ page = 81$
- Example 3
  - SELECT * FROM Skaters WHERE name < 'F%'
  - Assume around 10 000 tuples match(assume thataround 490 data pages)
  - $cost = 75\ leaf\ page + 490\ data\ pages = 565$
- Note: sorting expensive if leaf-pages do not fit in main-memory

Sort:

- Sometimes a pass 2 is needed
  - Pass 0 created more runs than there are main memory buffers
  - Therefore Pass 1 produces more than one run
  - Pass 2 takes the runs of Pass 1 and merges them
- Cost
  - SELECT sname, age FROM Skaters ORDER BY age
  - If everything fits into main memory (Only pass 0 needed):
    Read number of data pages
    sort and pipeline result into next operator (project)
  - Pass 0 + pass 1 needed
    Pass 0: read # pages, write # pages (have to write temp. results!)
    Pass 1: read # pages, sort and pipeline result into next operator
    3 * #pages
  - Pass 0 + pass1 + pass2 needed
    5 * #pages

Join cost estimation:

- $|Skaters\ Participates| = |Participates|$
  - Join attribute is primary key for Skaters
  - Each Participates tuple matches exactly with one Skaters tuple
- $|Skaters \times Participates| = |Participates| * |Skaters|$
  - Cross product is always the product of individual relation sizes
- For other joins more difficult to estimate (Continues in next episode…)

<table>
<tr>
<td>
<u>Nested loop joins:</u>
• Simple nested loop join: For each tuple in the outer relation P, we scan the entire inner relation S
- $cost = PartPages + CARD(P) * SkaterPages = 1000 + 100000 * 500$
• Page-oriented Nested Loops join: For each page of P, get each page of S, and write out matching pairs of tuples <p, s>, where p is in P-page and s is in S-page.
- $cost = PartPage + PartPage * SkaterPage = 1000 + 1000 * 500$
</td>
<td>
<u>Join cost on relation R1 and R2:</u>
Block oriented nested loop join:
• Smaller relation fits in main memory+2extra buffer page:
$cost = page(R1) + page(R2)$
• No relation fits in main memory(B join frame):
$cost = page(R1) + \frac{page(R2) * page(R1)}{B - 2}$
Index nested loop join
• Index on the join column of one of the relation(R2):
$cost = page(R1) + card(R1) * cost\_finding\_index(R2)$
• If the join attribute is primary key in inner relation
Sort merge join
• Sort P and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples
- Advance scan of P until current P-tuple >= current S tuple, then advance scan of S until current S-tuple >= current P tuple; do this until current P tuple = current S tuple.
- At this point, all P tuples with same value in Pi (current P group) and all S tuples with same value in Sj (current S group) match; output <p, s> for all pairs of such tuple 1s.

• P is scanned once; each S group is scanned once per matching P tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)
</td>
</tr>
</table>