Lec 2. Firebase rest & web api

CRUD in REST

CRUD

- C – create PUT/POST (verbs in HTTP)
- R – retrieve/read GET
- U – update PATCH
- D – delete DELETE

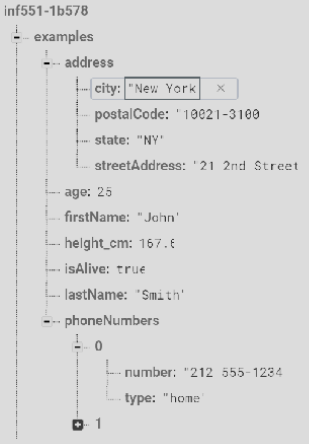
Get:

- `curl 'https://inf551-1b578.firebaseio.com/weather.json'`
- `curl -X GET 'https://inf551-1b578.firebaseio.com/weather.json'`

Another example

- curl -X GET 'https://inf551-1b578.firebaseio.com/examples/phoneNumbers/0.json'
  - {"number":"212 555-1234","type":"home"}

Note: refer to array element by index



PUT

`curl -X PUT 'https://inf551-1b578.firebaseio.com/weather.json' -d '"hot"'`
`– "hot"`
`• PUT: write a given value (e.g., "hot") to the specify node (e.g., "weather")`

- **Overwrite** if node already has value

`curl -X PUT 'https://inf551-1b578.firebaseio.com/users/100.json' -d '{"name": "john"}'`

- This will add a new node "users" (assuming it does not exist yet) and a child of this node with key `"100" and content: {"name": "john"}`

POST

`curl -X POST -d '{"name": "John"}' 'https://inf551-1b578.firebaseio.com/users.json`

- ★ **NOTE:** post automatically generates a **new key** & then stores the value for the new key
  - In contrast, **PUT** will simply **overwrite** the value

PATCH

`curl -X PATCH -d '{"name": "John Smith", "age": 25}' 'https://inf551-1b578.firebaseio.com/users/100.json'`

- ★ **NOTE:** patch expects an JSON object in the value
- PATCH performs the **update** if value already exists (e.g., name) ; otherwise, it inserts the new value (e.g., age)

DELETE

- `curl -X DELETE 'https://inf551-1b578.firebaseio.com/users/100.json'`
- `curl -X DELETE 'https://inf551-1b578.firebaseio.com/users.json'`

ordering and filtering

- orderBy="$key"
- orderBy="$value"
- orderBy="name"
- startAt=25, endAt=25, equalTo=25, equalTo="john"
- limitToFirst=1, limitToLast=1

Query: filtering by key

```
curl 'https://inf551-1b578.firebaseio.com/users.json?orderBy="$key"&equalTo="200"'
```

- Key must be a String

```
curl 'https://inf551-1b578.firebaseio.com/users.json?orderBy="$key"&startAt="200"
```

- Users with keys >= "200"

Ways of filtering data

By ***key***:

- orderBy="**$key**"

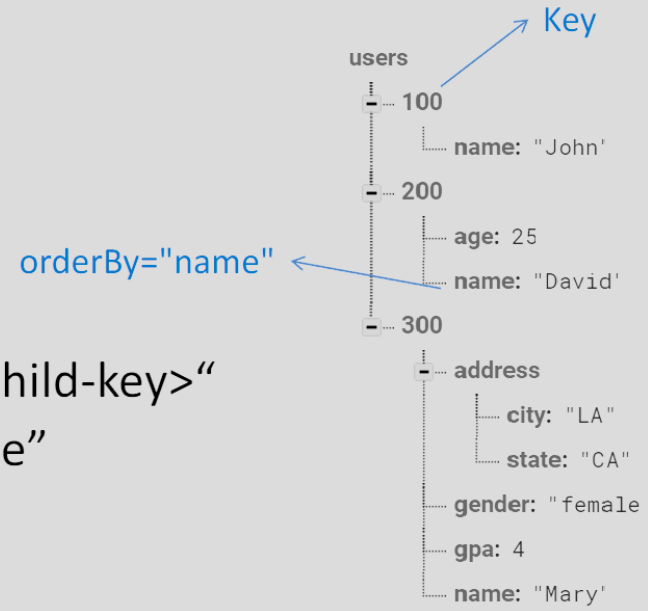
By ***child key***:

- orderBy="**<path-to-child-key>**"
-  : `orderBy = "name"`

By ***value***:

- orderBy="**$value**"

- By key:
  - orderBy="$key"
- By child key:
  - orderBy="<path-to-child-key>“
  - E.g., orderBy = “name”
- By value:
  - orderBy="$value"



Parameters

- startAt
- endAt
- equalTo
- limitToFirst
- limitToLast

🌰: filtering by *child key*

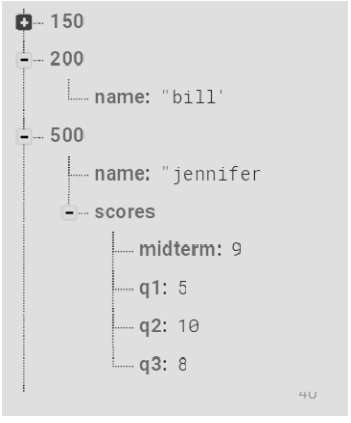
```
- curl 'https://inf551-1b578.firebaseio.com/users.json?orderBy="age"&limitToFirst=1&print=pretty'
```

🌰: orderBy="$value"

```
- curl -X PUT 'https://inf551-1b578.firebaseio.com/users/500.json' -d'{"name": "jennifer", "scores": {"q1": 5, "q2":10, "q3": 8, "midterm": 9}}'
```

🌰: filtering by value

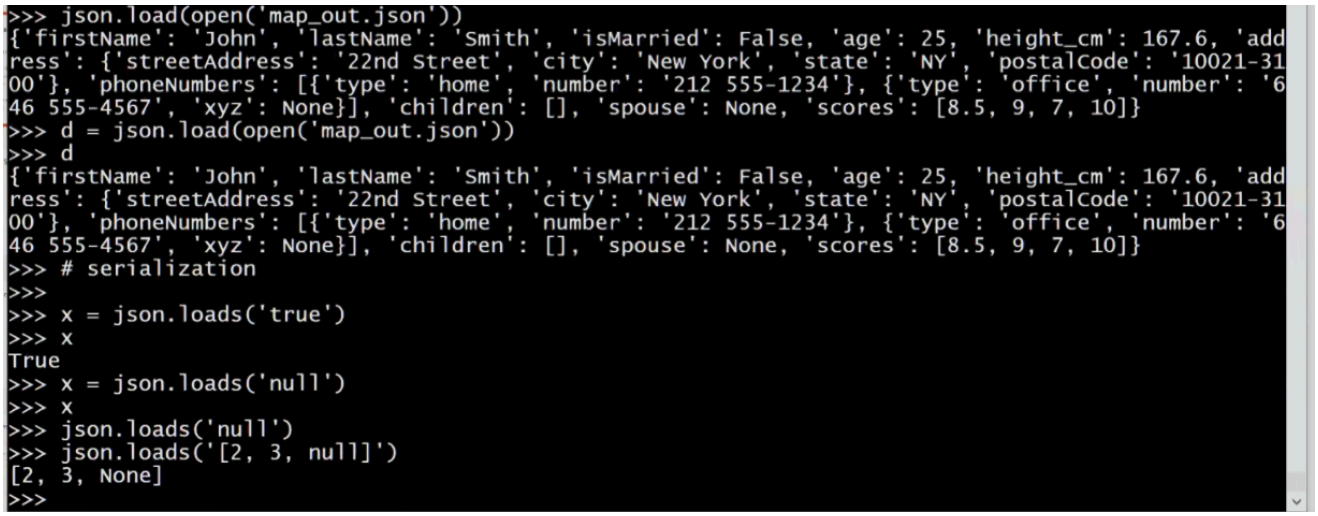
```
- curl 'https://inf551-1b578.firebaseio.com/users/500/scores.json?orderBy="$value"&limitToFirst=1&print=pretty'
```



⚠️**REST API 返回的结果未经排序**：JSON 解释程序不会强制对结果集进行排序。虽然 **`orderBy`** 可以与 **`startAt`**、**`endAt`**、**`limitToFirst`** 或 **`limitToLast`** 结合使用以返回数据的子集，但不会对返回的结果进行排序。因此，如果顺序很重要，您需要自己编写代码对结果进行排序。

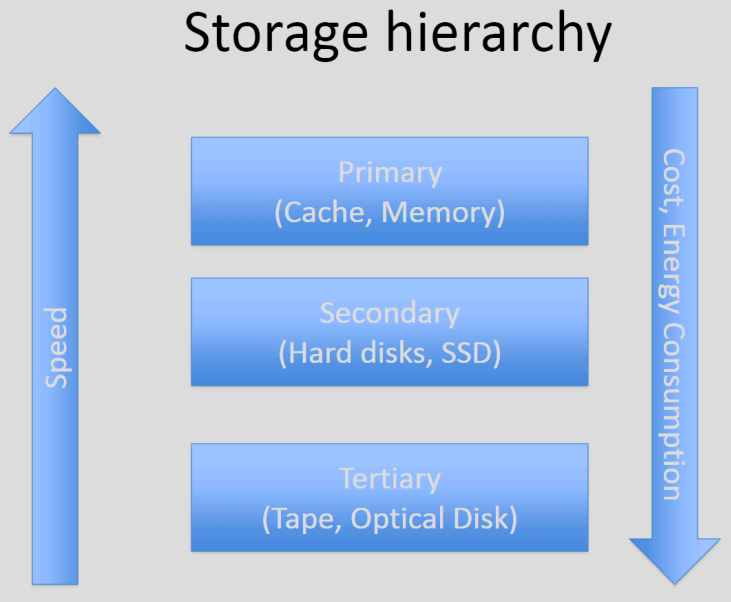
Syntax of JSON

- value = string|number|object|array|true|false|null
- object = {} | { *members* }
    - members = pair | pair, members
    - pair = string : value
- array = [] | [*elements*]
    - elements = value | value, elements



Lec3. Storage systems

Storage hierarchy



Access times

Time taken before drive is ready to transfer data

LEVEL	ACCESS TIME	TYPICAL SIZE
Registers	"instantaneous"	under 1KB
Level 1 Cache	1-3 ns	64KB per core
Level 2 Cache	3-10 ns	256KB per core
Level 3 Cache	10-20 ns	2-20 MB per chip
Main Memory	30-60 ns	4-32 GB per system
Hard Disk	3,000,000-10,000,000 ns	over 1TB

SSD: 25,000 ns

Ref:
https://arstechnica.com/information-technology/2012/06/inside-the-ssd-revolution-how-solid-state-disks-really-work/

Characterizing a storage device

- ***Capacity*** (bytes)
  - How much data it can hold
- Cost ($$$)
  - Price per byte of storage
- ***Bandwidth*** (bytes/sec)
  - Number of bytes that can be transferred per second
  - Note that read and write bandwidth may be different
- ***Latency*** (seconds)
  - Time elapsed, waiting for response/delivery of data

Time to complete an operation

- Time to complete an operation depends on both bandwidth and latency

$$CompletionTime = Latency + Size/Bandwidth$$

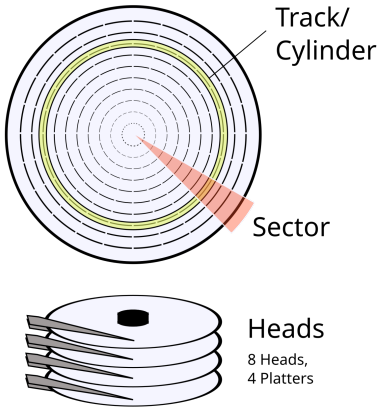
- The time for a workload may depend on
  - Technology, e.g., hard drive/SSD
  - Operation type, e.g., read/write
  - Number of operations in the workload
  - Access pattern (random vs. sequential)

Access pattern

- ***Sequentia***
  - Data to be accessed are located next to each other or sequentially on the device
- ***Random***
  - Access data located randomly on storage device

Disk organization

- Each platter consists of a number of ***tracks***
- Each track is divided into ***N*** fixed size ***sectors***
  - Typical ***sector size*** is 512 bytes (old) or **4KB** (new)
  - *Sectors* can be numbered from ***0*** to ***N-1***
  - **Entire** *sector* is written "atomically"
    - All or nothing



CHS (cylinder-head-sector)

- Early way to address a sector
  - Now LBA (Logical Block Addressing) more common

🌰 What is the capacity of the drive?

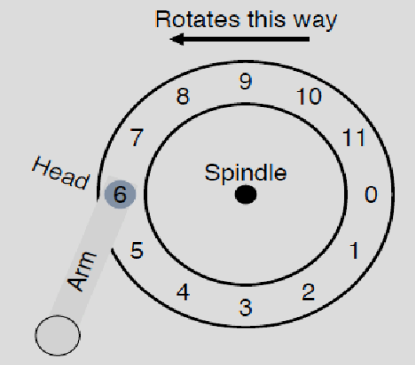
- # cylinders: 256 (= 2^8)
- # heads: 16 (i.e., 8 platters, 2 heads/platter) (= 2^4)
- # sectors/track: 64 (= 2^6)
- Sector size = 4KB (= 2^12)

- **the capacity of the drive =** $2^{8+4+6+12} = 2^{30}\ (bytes) = 1\ GB$

  = $\#\ cylinders\ *\ \#\ heads\ *\ \#\ sectors/track\ *\ Sector\ size$

Rotational latency

- Definition
  - Time for **waiting for the right sector to rotate under the head**
  - **On average: about 1/2 of time of a *full rotation***
    - Worst case?
    - Best case?



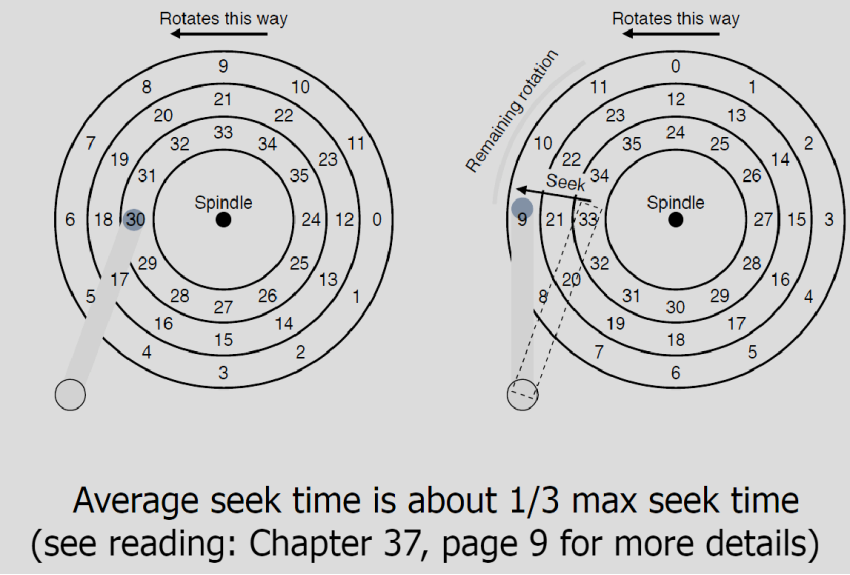
Rotation time

🌰: Assume 10,000 RPM (rotations per **minute**)

$$60,\ 000\ ms/\ 1,000\ rotations\ =\ 6\ ms\ /\ rotation$$

Multiple tracks: add seek times

- *Average* seek time is about **1/3** max seek time



Transfer time

- $Transfer\ time\ = data\_size\ /\ bandwidth$

1. Assume that we transfer 512KB.
2.Assume 128 MB/sec transmission bandwidth

$Transfer\ time\ = 512KB/128MB\ *\ 1,000ms\ =\ 4ms$

Completion time

- **Completion time:** $T_{Completion} = T_{seek} + T_{rotation} + T_{transfer}$
    - T_{seek}**: Time to get the disk head on right track**
        - Average seek time is about 1/3 max seek time
    - $T_{rotation}$**: Time to wait for the right sector to rotate under the head**
        - On average: about 1/2 of time of a *full rotation*
    - $T_{transfer}$**: Time to actually transfer the data**

- Capacity 4TB
- # platters: 4
- # heads: 8
- Bytes per sector: 4096

- Transmission bandwidth: 100MB/sec
- Maximum seek time: 12ms
- RPM: 10,000

Sector vs. block

- **Block has 1 or more sectors**
- Disk typically transfers one block at a time
- We will assume one block = one sector
    - Unless stated otherwise

Sequential operations

- May assume **all sectors involved are on same track**
    - We may need to seek to the right track or rotate to the ***first sector***
    - But **no rotation/seeking** needed afterward

Actual bandwidth

- Consider a workload ***w***
    - 🌰: ***w*** = sequential access of 100 blocks of data
    - Denote size (# of bytes) of data in ***w*** as |***w***|
    - 🌰: ***w*** = 400KB (100 blocks, 4KB/block)

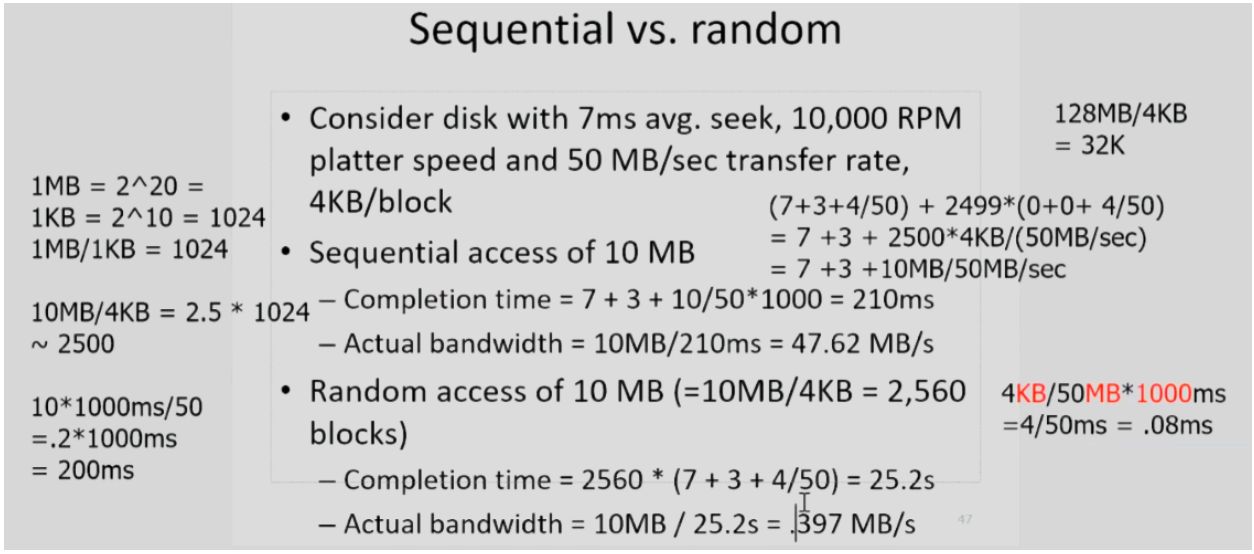
- Suppose completion time for: ***w*** = ***t***

- Actual bandwidth (with respect to ***w***): $|w| \ / \ t$

Sequential vs. random

🌰

- Consider disk with 7ms avg seek, 10,000 RPM platter speed and 50 MB/sec transfer rate, 4KB/block.
- Sequential access of 10 MB
    - Completion time = 7 + 3 + 10/50*1000 = 210ms
    - Actual bandwidth = 10MB/210ms = 47.62 MB/s
- Random access of 10 MB (2,500 blocks
    - Completion time = 2500 * (7 + 3 + 4/50) = 25.2s
    - Actual bandwidth = 10MB / 25.2s = .397 MB/s



SSD (Solid State Drives)

- Same form-factor and control interface as magnetic disks
- Significantly better latency
    - No seek or rotational delay
- Much better performance on random workload:
    - Benefits from improved latency
    - However, write has much higher latency (but see next slide)

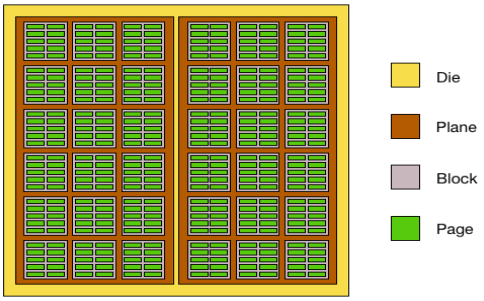
Writing to SSD is complicated

- Can not overwrite a page
    - Need to ***erase*** its block (at a certain point) instead

- *SSD controllers* take care of all these details

SSD

- Contains a number of flash memory chips
    - Chip -> dies -> planes -> blocks -> pages (rows) -> cells
        - Cells are made of floating-gate transistors
- ***Page*** is the smallest unit of data transfer between SSD and main memory
    - Much like a block in hard disk
    - Understanding Flash: Blocks, Pages and Program / Erases:



Dies, planes, block, and pages

- Typically, a *chip* may have 1, 2, or 4 dies
- A *die* may have 1 or 2 *planes*
- A *plane* has a number of *blocks*
    - *Block* is the smallest unit that can be erased
- A *block* has a number of *pages*
    - *Page* is the smallest unit that can be read, programmed/written

- Typical page and block sizes
    - Common page sizes: 2K, 4K, 8K, and 16K
    - A block typically has 128 to 256 pages
      => Block size: 256KB to 4MB

Read/write units

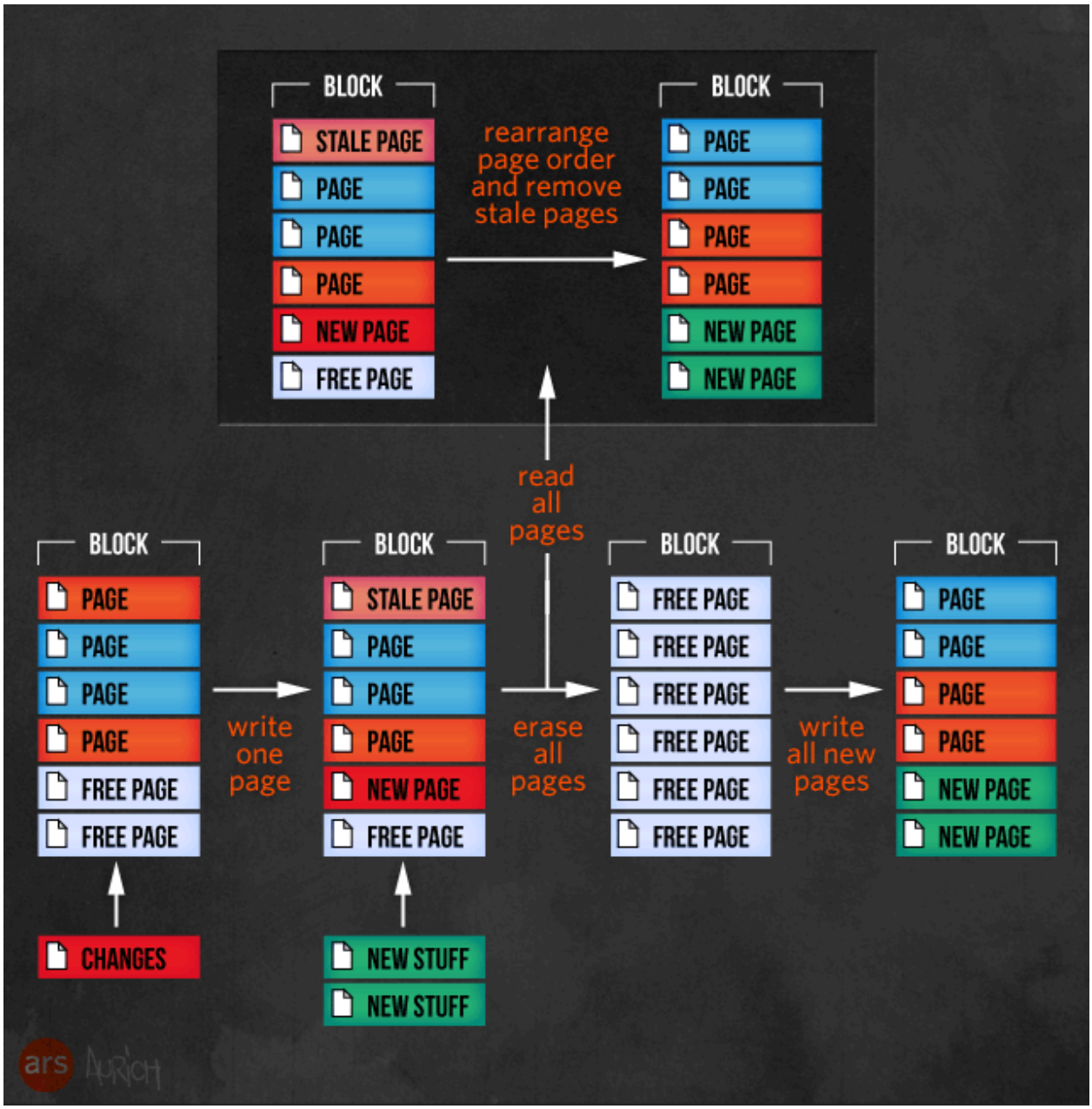
- ***Page*** is the smallest unit for read and write
    - (write is also called program, 1->0)
- ***Block*** is the smallest unit for erase (0->1)
    - i.e., make cells "empty" (i.e., no electrons)

Read/write units

- Page is the smallest unit for read and write (write is also called program, 1->0)
- Block is the smallest unit for erase (0->1)
  - i.e., make cells "empty" (i.e., no electrons)

Operation	Area
Read	Page
Program (Write)	Page
Erase	**Block**

[Solid-state revolution: in-depth on how SSDs really work]



P/E cycle

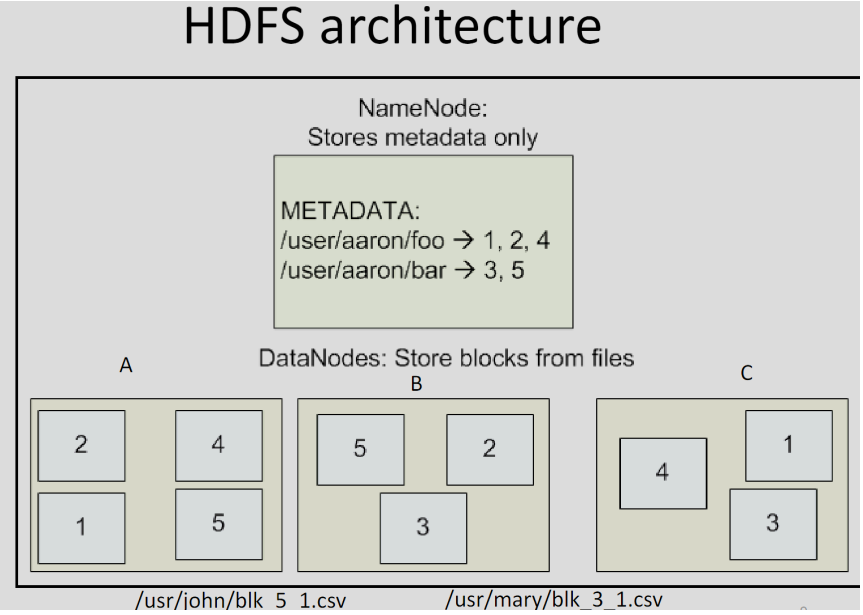
- *P*: *program*/*write*; *E*: *erase*
- Every write & erase damages oxide layer surrounding the floating-gate to some extent
- P/E cycle:
    - Data are written to cells (P): cell value from 1 -> 0
    - Then erased (E): 0 -> 1

Lec 4. Hadoop HDFS

- HDFS (Hadoop distributed file system)
    - Distributed data storage with high reliability
- MapReduce
    - A parallel, distributed computational paradigm
    - With a simplified programming model

HDFS

- Data are distributed among multiple data nodes
    - Data nodes may be added on demand for more storage space
- Data are replicated to cope with node failure
    - Typically replication factor: 2 or 3
- Requests can go to any replica
    - Removing the bottleneck (as in single file server)



HDFS has ...

- A single ***NameNode***, storing meta data:
  - A hierarchy of directories and files (name space)
  - Attributes of directories and files (in inodes), e.g., permission, access/modification times, etc.
  - Mapping of files to blocks on data nodes
- A number of ***DataNodes***:
  - Storing contents/blocks of files

- Compute nodes
  - Data nodes are compute nodes too
  - Advantage:
    - Allow schedule computation close to data

- A SecondaryNameNode
  - Maintaining checkpoints/images of NameNode
  - For recovery
  - not a fail over node
- In a single-machine setup
  - all nodes correspond to the same machine

Metadata in NameNode

- NameNode has an ***inode*** for each file and dir
- Record attributes of file/dir such as
  - Permission
  - Access time
  - Modification time
- Also record mapping of files to blocks
  - 每个inode保存了文件系统对象数据的属性和磁盘块位置

- Mapping information in NameNode

  🌰: file `/user/aaron/foo` consists of blocks 1, 2, and 4
  Block 1 is stored on data nodes 1 and 3
  Block 2 is stored on data nodes 1 and 2

Block size

- HDFS: 128 MB (version 2 & above)
  - Much larger than disk block size (4KB)
  - A: 128MB; B: 4KB
  - 128MB/4KB = 32K
  - A: 1GB/128MB = 8; B: 1GB/4KB = 2^30/2^12 =2^18 = 2^8K = 256K

- Why larger size in HDFS?
  - Reduce metadata required per file
  - Fast streaming read of data (since larger amount of data are sequentially laid out on disk)

HDFS

- HDFS exposes the concept of blocks to client

- Reading and writing are done in two phases
  - Phase 1: client asks *NameNode* for block locations
    - By calling (sending request) getBlockLocations(), if reading
    - Or calling addBlock() for allocating new blocks (one at a time), if writing (need to call create()/append() first)
  - Phase 2: client talks to *DataNode* for data transfer
    - Reading blocks via readBlock() or writing blocks via writeBlock()

Key operations

- Reading:
  - **`getBlockLocations()`**
- Writing
  - **`create()`**
  - **`append()`**
  - **`addBlock()`**

`getBlockLocations()`

- Before reading, client needs to first obtain locations of blocks
- Input:
  - File name
  - Offset (to start reading)
  - Length (how much data to be read)
- Output:
  - Located blocks (data nodes + offsets)

Creating a file

- Needs to specify:
  - Path to the file to be created, e.g., /foo/bar
  - Permission mask
  - Client name
  - Flag on whether to overwrite (entire file!) if already exists
  - How many replicas
  - Block size

```
 */
@AtMostOnce
HdfsFileStatus create(String src, FsPermission masked,
    String clientName, EnumSetWritable<CreateFlag> flag,
    boolean createParent, short replication, long blockSize,
    CryptoProtocolVersion[] supportedVersions)
    throws IOException;
```

Creating a new file

Operations

- `readBlock()`
- `writeBlock()`
- `copyBlock()` – for load balancing
- `replaceBlock()` – for load balancing
  - Move a block from one DataNode to another

Reading a file

1. Client first contacts NameNode which informs the client of the **closest *DataNodes*** storing blocks of the file
   a. This is done by making which RPC call?
2. Client contacts the DataNodes directly for reading the blocks
   a. Calling readBlock()

Writing a file

- Blocks are written one at a time
  - In a pipelined fashion through the data nodes
- For each block:
  - Client asks NameNode to select DataNodes for holding its replica (*using which rpc call?*)
    🌰: DataNodes 1 and 3 for the first block of `/user/aaron/foo`
  - It then forms the pipeline to send the block

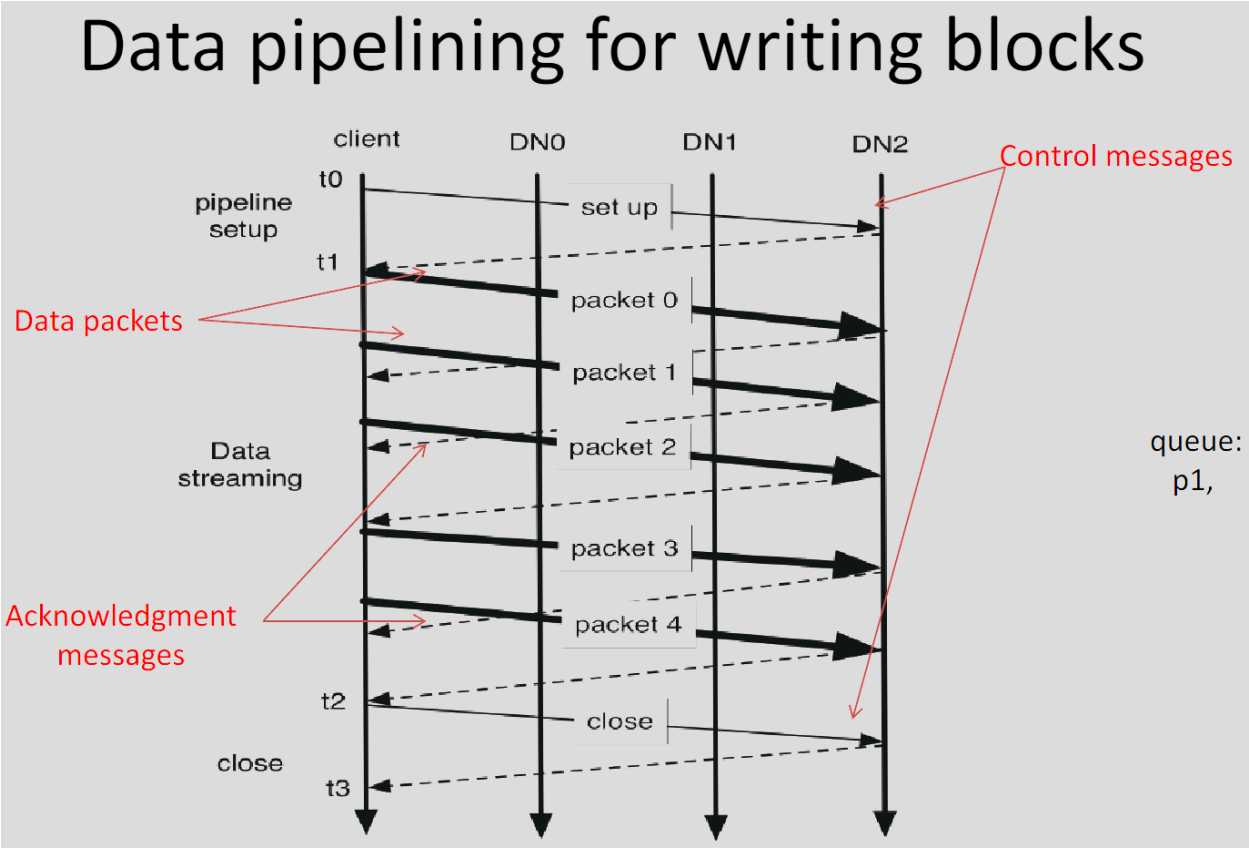
Data pipelining

- Consider a block X to be written to DataNode A, B, and C (replication factor = 3)

1. X is broken down into packets (typically 64KB/packet)
    - 128MB/64KB = 2048
2. Client sends the packet to DataNode A
3. A sends it further to B & B further to C

Acknowledgement

- Client maintains an ack (acknowledgment) queue
- Packet removed from ack queue once received by all data nodes
- When all packets were written, client notifies NameNode
    - NameNode will update the metadata for the file
    - Reflecting that a new block has been added to the file



Lec 5. File systems

File and directory

- When creating a file
  - Bookkeeping data structure (inode) created:
    - recording size of file, location of its blocks, etc.
  - Linking a human-readable name to the file
  - Putting the link in a directory

Metadata: Info about file (stored in inode)

Info about file (stored in inode)

```
struct stat {
    dev_t st_dev; /* ID of device containing file */
    ino_t st_ino; /* inode number */
    mode_t st_mode; /* protection */
    nlink_t st_nlink; /* number of (hard) links */
    uid_t st_uid; /* user ID of owner */
    gid_t st_gid; /* group ID of owner */
    dev_t st_rdev; /* device ID (if special device file, e.g., /etc/tty) */
    off_t st_size; /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t st_blocks; /* number of blocks allocated */
    time_t st_atime; /* last time file content was accessed */
    time_t st_mtime; /* last time file content was modified */
    time_t st_ctime; /* last time inode was changed */
};
```

- Execute "man -S 2 stat" for more details...

inode

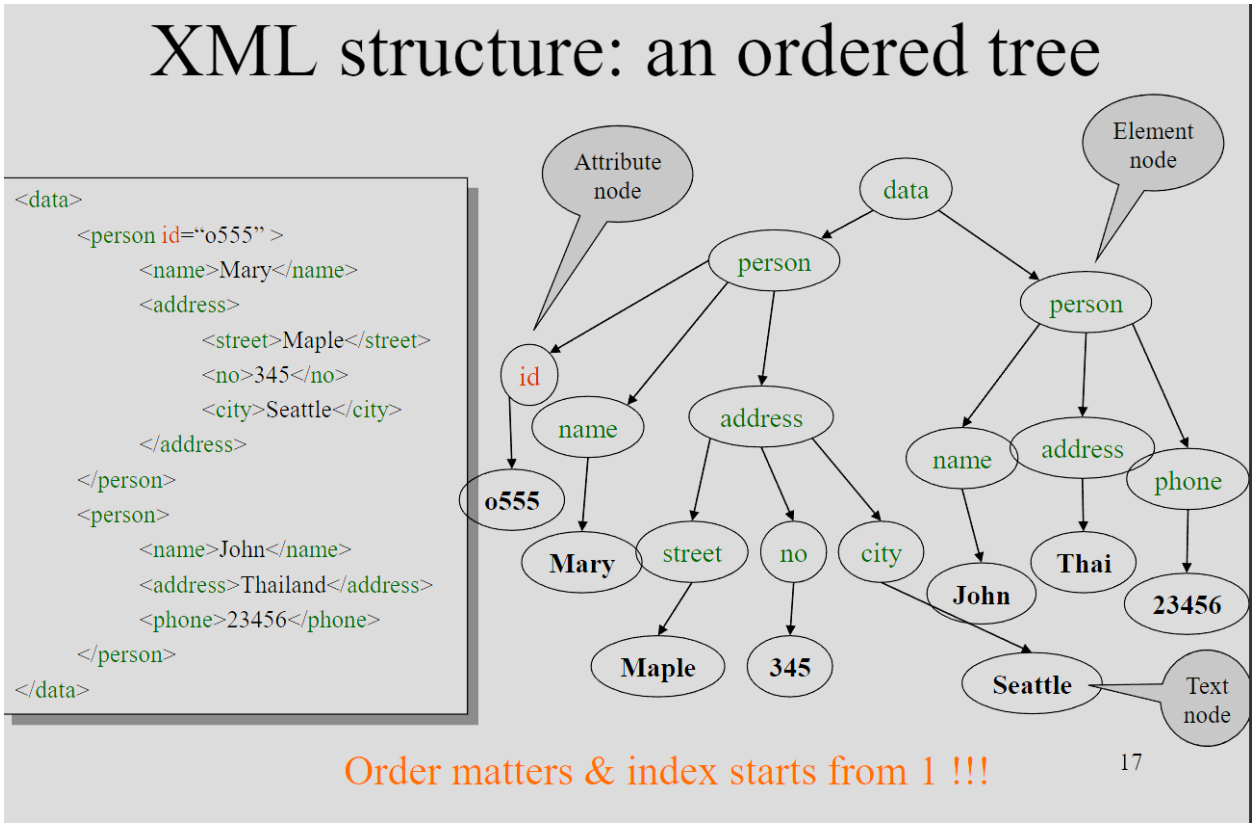
- Stores metadata/attributes about the file
- Also stores locations of blocks holding the content of the file

Lec 6. XML

Ajax

- ***A***synchronous ***J***avascript ***a***nd ***X***ML
- Web clients send and receive data from server asynchronously
    - Benefit: more responsive web pages
    - Common to use XML, JSON as data format

XML structure: an ***ordered tree***



XML Data

- XML is self-describing
- Schema elements become part of the data
    - Relational schema: person(name, phone)

    - In XML <persons>, <name>, <phone> are part of the data, and are repeated many times
    - Consequence: XML is much more flexible
    - XML = semi-structured data

DTD（Document Type Definitions）

- A set of markup for describing schema of XML data
- an XML document may have a DTD
- XML document:
    - **well-formed** = *if tags are correctly closed*
    - **valid** = *if it has a DTD and conforms to it*
- validation is useful in data exchange

Querying XML Data

- **XPath**
    - simple navigation through the tree
- **XQuery**
    - the SQL of XML

Lec 7. ER model

箭头→ vs 圆箭头 –)

    - (1:N) vs (0:N)

- 箭头 →:
    - 箭头所指 entity 为 ”多”
    - 箭头出发 entity 为 (0 : 1)，至多有一个对应存在
- 圆箭头 –):
    - 箭头所指 entity 为 ”多”
    - 箭头出发 entity 为 1，至少有一个对应存在（=exactly one）

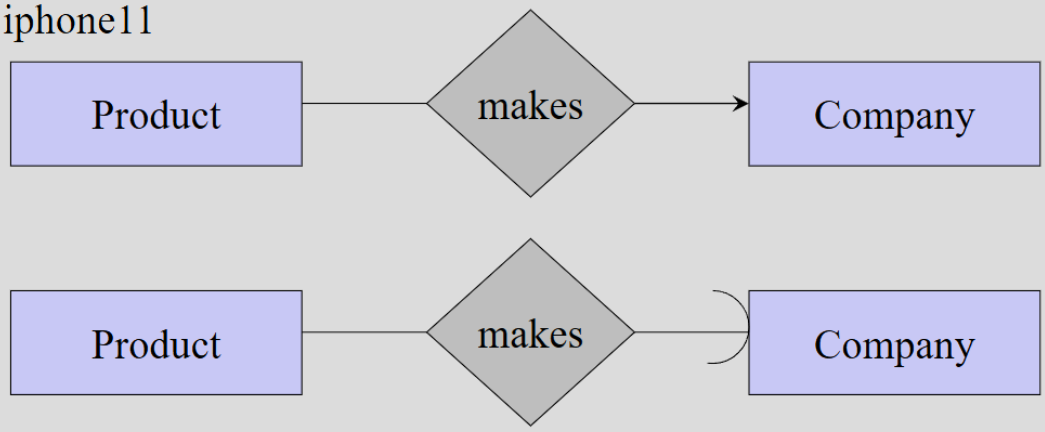
    - iPhone 99 may exist in Product but would not have a corresponding Company
      So. should use “箭头 →”
    - 如果能确定一定存在对应，根据情况可以删掉makes relation，将其整合进Product

Referential Integrity Constraints

Product(id, tite)
100, iphone11

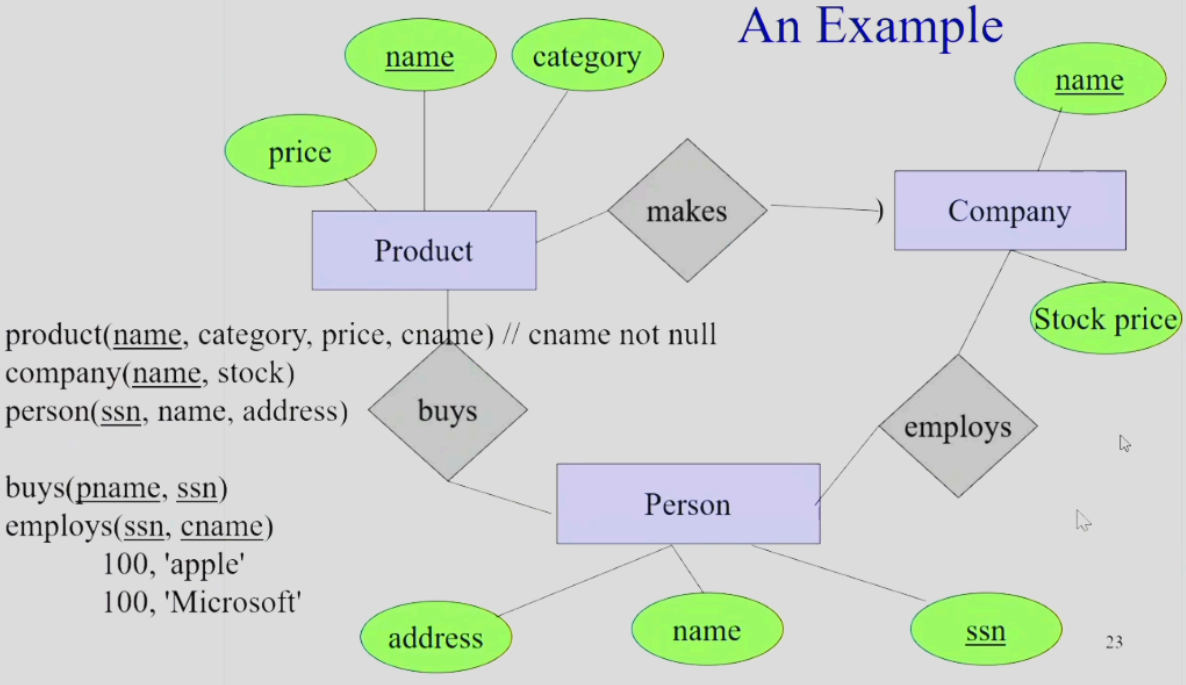
Makes(pid, cname)

Company(name, addr)



- This will be even clearer once we get to relational databases

An Example



Lec 8. SQL (SQL 语法)

数据库中的内连接、自然连接、和外连接的区别 (link)

数据中的连接join分为内连接、自然连接、外连接，外连接又分为左外连接、右外连接、全外连接

表1		
A	B	C
1	2	3
5	6	7

表2		
C	D	E
3	4	5
8	9	1

当然，这些分类都是在连接的基础上，是从两个表中记录的笛卡尔积中选取满足连接的记录。笛卡尔积简单的说就是一个表里的记录要分别和另外一个表的记录匹配为一条记录，即如果表A有3条记录，表B也有三条记录，经过笛卡尔运算之后就应该有3*3即9条记录。如下表：

A	B	C	C	D	E
1	2	3	3	4	5
5	6	7	8	9	1
1	2	3	8	9	1
5	6	7	3	4	5

自然连接（natural join）

自然连接是一种特殊的等值连接，他要求两个关系表中进行比较的必须是相同的属性列，无须添加连接条件，并且在结果中消除重复的属性列。

sql语句：Select …… from 表1 natural join 表2

结果：

A	B	C	D	E
1	2	3	4	5

内连接（inner join）

内连接基本与自然连接相同，不同之处在于自然连接要求是同名属性列的比较，而内连接则不要求两属性列同名，可以用using或on来指定某两列字段相同的连接条件。

sql语句：Select …… from 表1 inner join 表 2 on 表1.A=表2.E

结果：

A	B	1.C	2.C	D	E
5	6	7	3	4	5

左外连接（left outer join)

左外连接是在两表进行自然连接，只把左表要舍弃的保留在结果集中，右表对应的列上填null。

sql语句：Select …… from 表1 left outer join 表2 on 表1.C=表2.C

结果：

A	B	C	D	E
1	2	3	4	5
5	6	7	null	null

右外连接(rignt outer join)

右外连接是在两表进行自然连接，只把右表要舍弃的保留在结果集中，左表对应的列上填null。

Select …… from 表1 rignt outer join 表2 on 表1.C=表2.C

结果：

A	B	C	D	E
1	2	3	4	5
null	null	8	9	1

全外连接(full join)

全外连接是在两表进行自然连接，只把左表和右表要舍弃的都保留在结果集中，相对应的列上填null。

Select …… from 表1 full join 表2 on 表1.C=表2.C

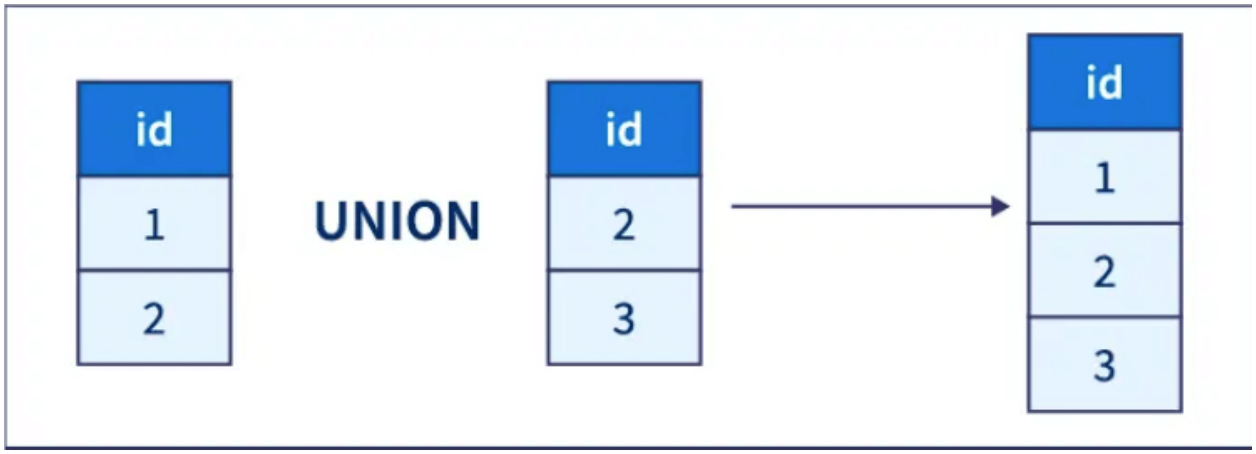
结果：

A	B	C	D	E
1	2	3	4	5
5	6	7	null	null
null	null	8	9	1

SQL UNION 操作符 (重点在 '列')

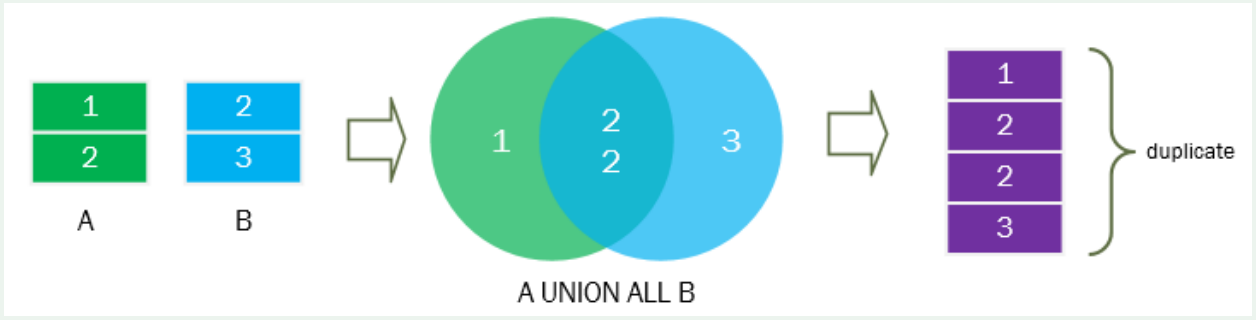
- SQL UNION 操作符合并两个或多个 SELECT 语句的结果。
- UNION 操作符用于合并两个或多个 SELECT 语句的结果集。它可以从多个表中选择数据，并将结果集组合成一个结果集。使用 UNION 时，每个 SELECT 语句必须具有相同数量的列，且对应列的数据类型必须相似。

UNION 操作符默认会去除重复的记录，如果需要保留所有重复记录，可以使用 UNION ALL 操作符。



SQL UNION ALL 语法

```
SELECT column1, column2, ...
FROM table1
UNION ALL
SELECT column1, column2, ...

FROM table2;
```



Lec 11. query-execution

Cost Model

Cost parameters

- **M** = number of blocks/pages that are available in main memory
- **B(R)** = number of blocks holding R
- **T(R)** = number of tuples in R
- **V(R,a)** = number of distinct values of the attribute a of R

Estimating the cost of physical operators:

- Important in query optimization
- Here we consider I/O cost only
- We assume operands are relations stored on disk, but operator results will be left in main memory (e.g., pipelined to next operator in query plan)
- So we don't include the cost of writing the result

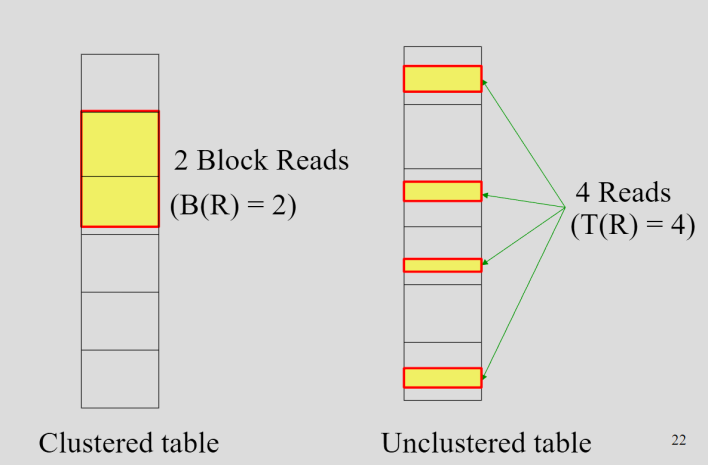
I/O Cost

- # of blocks read from or written to disk
- Recall that disk reads/writes data in the unit of block

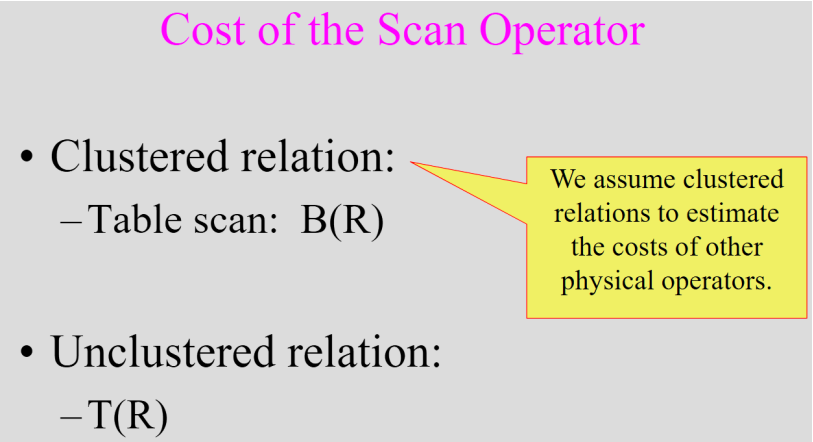
Scanning Tables (clustered / unclustered)

- Reading every row of tables
- The table is *clustered* (i.e., block consists only of records from this table):
  - # of I/O's = # of blocks
- The table is *unclustered* (e.g. its records are placed in blocks with those of other tables)
  - May need one block read for each record

Scanning Clustered/Uncluserted Tables



Cost of the Scan Operator



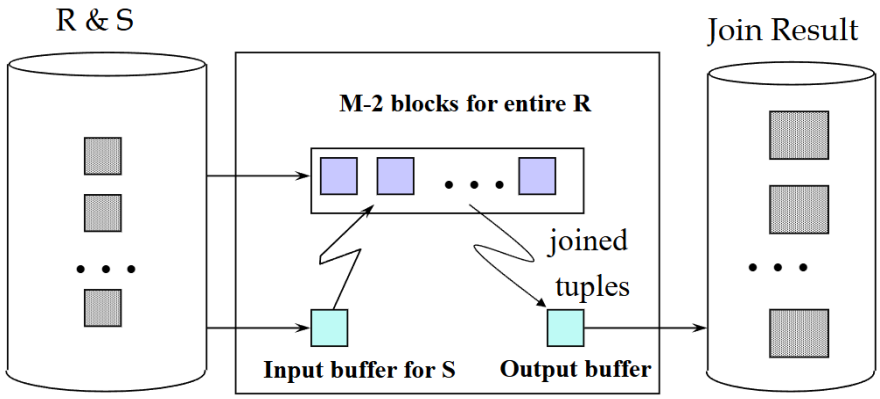
Classification of Physical Operators

- **One-pass algorithms**
  - Read the data only once from disk
  - Usually, require at least one of the input relations fits in main memory
- **Nested-Loop Join algorithms (1.x)**
  - Read one relation only once, while the other will be read repeatedly from disk
- **Two-pass algorithms**
  - First pass: read data from disk, process it, write it to the disk
  - Second pass: read the data for further processing
- **K-pass algorithms**
  - If data are too big or memory is too small, the algorithm may need k > 2 passes over the data

One-pass Algorithms

Binary operations: R ∩ S, R U S, R – S, R ⋈ S

- Assumption: min(B(R), B(S)) <= M (or M-2 to be exact)
- Scan a smaller table of R and S into main memory, then read the other one, block by block
- Cost: B(R)+B(S) (assume both are clustered)
- E.g. R ∩ S (assume set-based, no duplicates)
  - Read S into M-2 buffers and build a search structure
  - Read each block of R, and for each tuple t of R, see if t is also in S.
  - If so, copy t to the output; if not, ignore t



M = 102
B(R) <= 100

Nested-loop join (none of tables fits in memory...)

Tuple-based Nested Loop Joins

Tuple-based Nested Loop Joins

- Join R ⋈ S
- Assume neither relation is clustered

```
for each tuple r in R do
  for each tuple s in S do
    if r and s join then output (r,s)
```

- Cost: T(R) T(S)

Block-based Nested Loop Joins

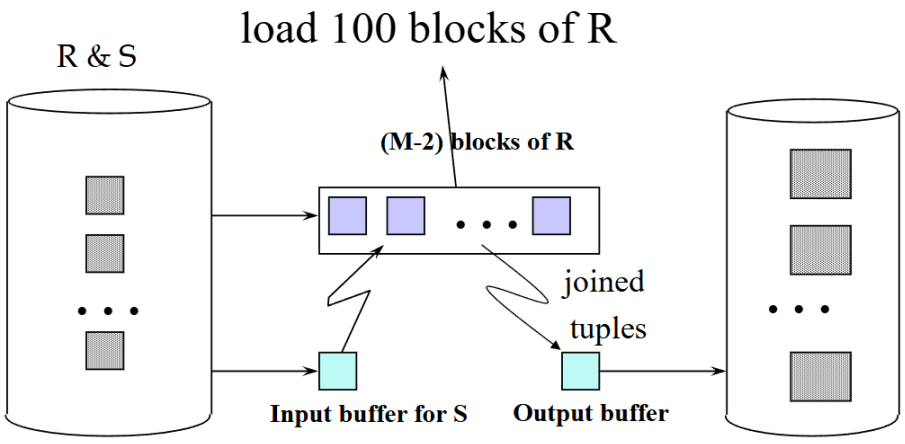
Block-based Nested Loop Joins

- Assume both relations are clustered

```
for each (M-2) blocks b_r of R do
  for each block b_s of S do
    for each tuple r in b_r do
      for each tuple s in b_s do
        if r and s join then output(r,s)
```

- Assume B(R) ≤ B(S) & B(R) > M

Block-based Nested Loop Joins



cost(R is outer) = B(R) + B(R)/(M-2) * B(S)
cost(S is outer) = B(S) + B(S)/(M-2) * B(R)

Notes

B(R) = 1000 B(S) = 5000, M = 102 (smallest M = 3)

- load 1st 100 blocks of R (R1), join with S (one block a time)
  100 + B(S)
- load 2nd 100 blocks of R, join with S
  100 + B(S)

…

- load 10th 100 blocks, join with S

R is outer:
* Total cost: B(R) + B(R)/(M-2) * B(S)
= 1000 + 10 * 5000 = 1000 + 50,000

S is outer:
* Total cost: B(S) + B(S)/(M-2) * B(R)

lower cost if smaller relation (R) placed in the outer

Example

- Suppose M = 102 blocks (i.e., pages), B(R) = 1000 blocks, B(S) = 5,000 blocks
  - # of chunks from R = 10, chunk size = 100 blocks

- Cost of R ⋈ S using block-based nested-loop join algorithm
  - If R is outer relation: one pass R; 10 passes through S
    - 1000 blocks + 1000/(102-2) * 5000 = 51,000
  - If S is outer relation: one pass S; 50 passes R
    - + 5000/(102-2) * 1000 = 55,000

Two-pass algorithms

- If an operation can not be completed in one pass, can we design an algorithm to complete it in two passes?
  - Yes, but with certain restriction on the relation size

Ideas

- Sorting
  - Sort relation(s) into **runs**
  - Perform the needed operation while merging the runs
- Hashing
  - Hash relation(s) into **buckets**
  - Only need to examine a bucket or a pair of buckets at a time

⭐Pass vs. Run vs. Bucket

以一个具体例子来说明这三个概念的区别：

假设我们有 100GB 的数据需要处理，但只有 1GB 的内存：

1. Pass（遍）
   - 指完整扫描数据集的一次过程
   - 例如：第一遍读取 100GB 数据，第二遍再读取 100GB 数据
   - 用于描述算法需要多少次完整扫描

2. `Runs`(有序子序列)
    - 在排序算法中使用
    - 比如将 100GB 数据分成 100 个 1GB 的块，每块内部排序后形成一个 run
    - 每个 run 内部是有序的，但 run 之间可能是无序的
3. `buckets`(桶)
    - 在哈希算法中使用
    - 例如:用哈希函数将 100GB 数据分到 100 个 bucket 中
    - 相同哈希值的数据会进入同一个 bucket
    - bucket 内部不保证有序

主要区别:

- `pass` 描述的是处理的次数
- `runs` 是排序算法的中间结果
- `buckets` 是哈希算法的中间结果

使用场景:

- 排序优先算法会产生 runs
- 哈希优先算法会产生 buckets
- 两种算法都可能需要多次 pass

总的来说，`pass` 是描述算法扫描数据的次数，而 `runs` 和 `buckets` 是两种不同算法在处理数据时产生的中间结构。

Duplicate Elimination (R) Based on Sorting

Duplicate Elimination δ(R) Based on Sorting

- Simple idea: sort first, then eliminate duplicates
- Pass1: sort runs of size M, write
  - Cost: 2B(R)
- Pass 2: merge M-1 runs, but include each tuple only once
  - Cost: B(R)

- Total cost: 3B(R), Assumption: $B(R) \leq M^2$
  - since B/M = # of runs
  - # of runs has to be <= M-1 to complete the merging in the second pass
  - So $B/M \leq M - 1$

48

Q. 所以在在第二遍(Pass 2)，我们需要把所有的runs都加载进内存吗？

A. 不需要把所有 runs 都完整加载进内存。我们只需要：

1. 为每个 run 分配一个缓冲区(buffer)，用于读取该 run 的当前数据块
2. 分配一个输出缓冲区，用于存储合并结果

具体工作方式：

```
假设有3个runs，每个run都很大，但我们只需要：

- Run1的buffer: [1,3,5] （从Run1读取几个数据）

- Run2的buffer: [2,4,6] （从Run2读取几个数据）

- Run3的buffer: [1,3,5] （从Run3读取几个数据）

- Output buffer: [] （用于存储合并结果）
```

当buffer中的数据用完时，再从对应的run中读取下一块数据。这样：

1. 每个 run 只需要一个较小的 buffer
2. 内存中同时存在 M-1 个输入 buffer（因为要合并 M-1 个runs）
3. 一个输出 buffer
4. 总内存使用量远小于数据总量

这就是为什么原文中说需要满足条件：runs 的数量 ≤ M-1，因为我们需要有足够的内存来容纳这些 buffer。

Q. $B(R) \leq M^2$?

推导过程:

- B/M = runs 的数量 (因为每个 run 大小为 M)
- runs 的数量必须 ≤ M-1 才能在第二遍中完成合并
- 因此 B/M ≤ M-1
- 也就是说 $B(R) \leq M^2$

Binary operations: R ∩ S, R U S, R – S Based on Sorting

- Idea: sort R, sort S, then do the right thing
- A closer look:
  - Step 1: split R into runs of size M, then split S into runs of size M. Cost: 2B(R) + 2B(S)
  - Step 2: merge M-1 runs from R and S; output a tuple on a case by cases basis
- Total cost: 3B(R)+3B(S)
- Assumption: $B(R)+B(S) \leq M^2$

Join Operation

Problem with join

- A large number of tuples with the same value on the join attribute(s)
- But buffer can not hold all joining tuples (with the same value on join attribute) for at least one relation

Simple sort vs. Sort-merge

notes (simple-sort)

1. completely sort R:
   – R (1000) => 10 runs => 1 run
   – cost: 4B(R)
2. completely sort S:
   – S (50,000) => 500 runs => 5 runs => 1 run
   – cost: 6B(S)
3. merge R and S (both sorted)
   - cost: B(R) + B(S)

Total cost: 5B(R) +7B(S)

Notes (sort-merge)

- R (1000 blocks) => 10 runs
  – cost: 2 B(R)
- S (50,000) => 500 runs => 5 runs
  – cost: 4 B(S)
- join by merging 10 runs with 5 runs
  – cost: B(R) + B(S)
- total: 3B(R) + 5B(S)

simple sort将所有Block进行sort直到 → 1 run

- 会多出2个 I/O operation

Main difference between the sort-merge join and simple sort-based join algorithm:

- The main difference is that we completely sort the relations (1 run for each) before we do the join in ***simple sort-based join***.
- However, in ***sort-merge join***, we start the join as soon as the total runs of two relations can fit in the memory buffers.

1. **Sort-Merge** Join
   - Assume buffer is enough to hold join tuples for at least one relation
     - Note that buffer also needs to hold a block for each run of the other relation

   - Total cost: 3B(R)+3B(S)
   - Assumption: $B(R)+B(S) \leq M^2$

Example

- Suppose M = 101 blocks (i.e., pages), B(R) = 1,000 blocks, B(S) = 5000 blocks (R.a = S.a)
  - Suppose we use 100 blocks in sorting

- Cost of R ⋈ S using sort-merge join algorithm
  - Pass 1: sort R => 10 runs, 100 blocks/run
    sort S => 50 runs, 100 blocks/run
  - Pass 2 (merge): B(R) + B(S)
  - total cost: 3B(R) + 3B(S) => 3B(R) + 3B(S)
- What if B(S) = 50,000 blocks?
  - 3B(R) + 5B(S)

Notes

- M = 101, B(R) = 1000, B(S) = 5000
  - R.a = S.a
- sort R =>10 runs, cost = 2*B(R)
- sort S => 50 runs, cost = 2*B(S)
- merge 10 runs from R with 50 runs from S
  - cost = B(R) + B(S)
- cost:
  - 3* B(R) + 3* B(S)

2. **Simple Sort**-based Join

Simple Sort-based Join

- Start by **completely** sorting both R and S on the join attribute (assuming this can be done in 2 passes):
  - Cost: `4B(R)+4B(S)` (because we need to write result to disk)
- Read both relations in sorted order, match tuples
  - Cost: `B(R)+B(S)`
- Can use as many buffers as possible to load join tuples from one relation (with the same join value), say R
  - Only one buffer is needed for the other relation, say S
- If we still can not fit all join tuples from R
  - Need to use nested loop algorithm, higher cost
- Total cost: 5B(R)+5B(S)
- Assumption: B(R) <= M^2, B(S) <= M^2, and at least one set of the tuples with a common value for the join attributes fit in M (or M-2 to be exact)
  - Note that we only need one page buffer for the other relation

Example

- Suppose M = 101 blocks (i.e., pages), B(R) = 1,000 blocks, B(S) = 5,000 blocks
  - Assume that we use 100 blocks for sorting pass

- Cost of R ⋈ S using simple sort-based join algorithm
  - Sort R (completely) => R': 4B(R) = 4000
  - Sort S => S': 4B(S) = 20,000
  - Join by merging R' with S': B(R) + B(S) (loading)
- What if B(S) = 50,000 blocks?
  - 500 runs => 5 runs => 1 run

Notes

- M = 101 (but 100 for sorting)
- B(R) = 1000 blocks

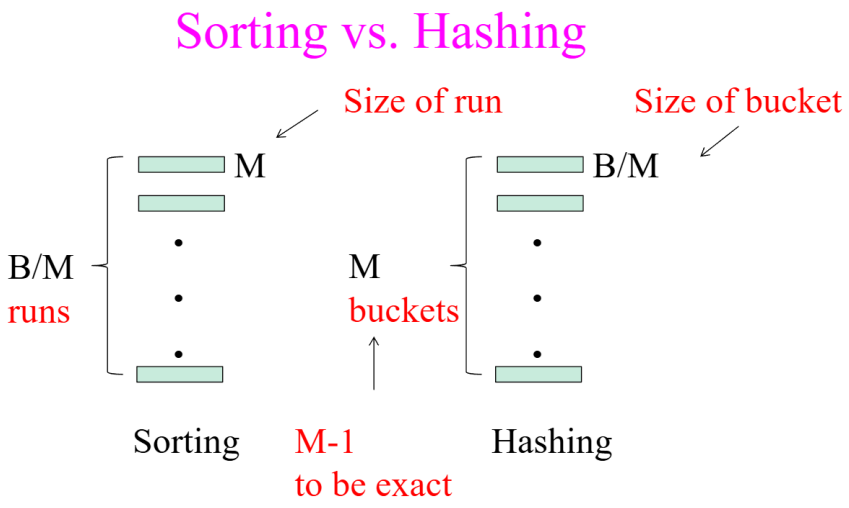
- completely sort R => R':
  - pass 0: load 100 blocks of R at a time => 10 runs
  - pass 1: merge 10 runs into a single run
  - cost: 2 * 2 * 1000 = 4000 or 4B(R)
- completely sort S => S':
  - cost: 4B(S)

Two-Pass Algorithms Based on Hashing

Hashing-Based Algorithms

- Hash all the tuples of input relations using an appropriate hash key such that:
  - All the tuples that need to be considered together to perform an operation go to the same bucket
- Reduce the size of input relations by a factor of M
- Perform the operation by working on a bucket (or a pair of buckets for binary operations) at a time
  - Apply a one-pass algorithm for the operation

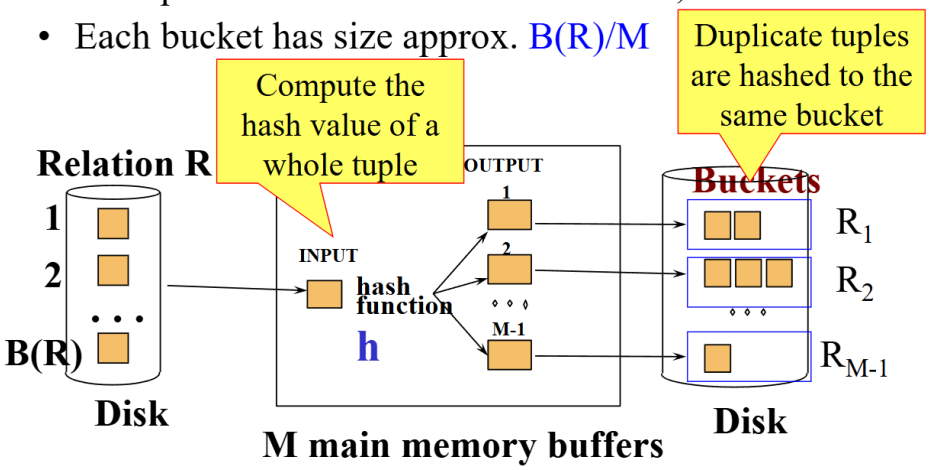
Sorting vs. Hashing



"Partitioning" picture

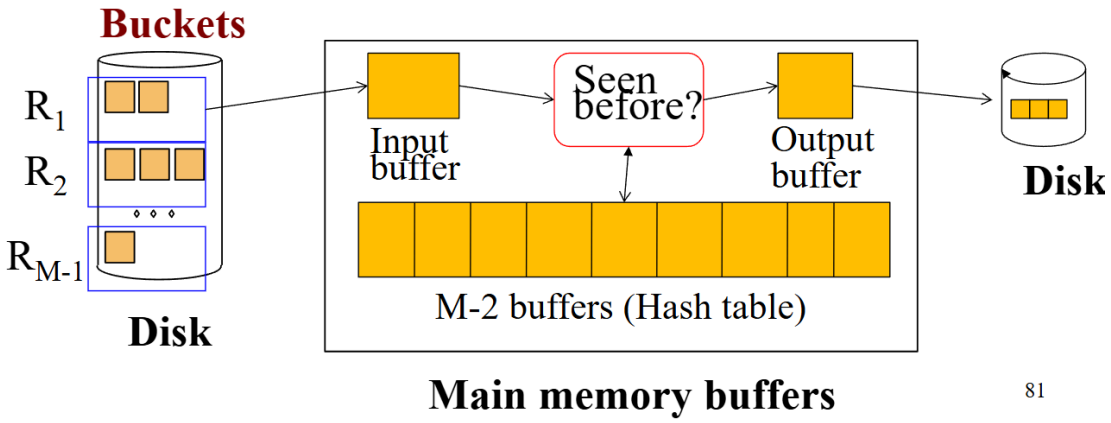
Two-Pass Duplicate Elimination Based on Hashing

- Idea: partition a relation R into buckets, on disk
- Each bucket has size approx. B(R)/M



Two Pass Duplicate Elimination Based on Hashing

- Does each bucket fit in main memory ?
  - Yes if B(R)/(M-1) <= M-2 (i.e., approx. B(R) <= M^2)
- Apply the one-pass δ algorithm for each R_i



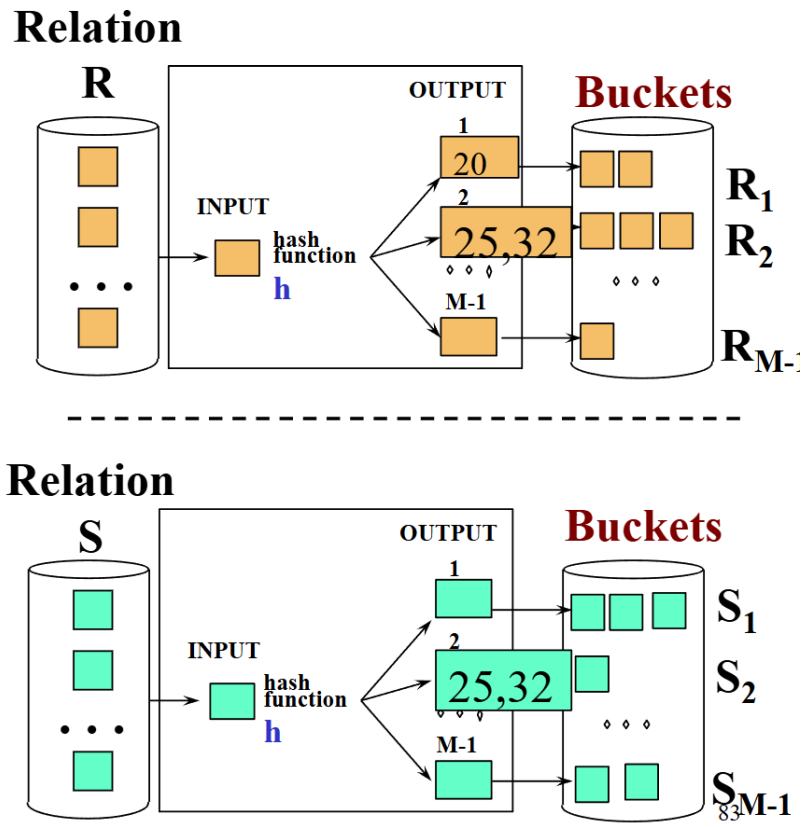
Partitioned Hash Join

R ⋈ S

- Step 1.a:
  - Hash S into M – 1 buckets
  - send all buckets to disk
- Step 1.b
  - Hash R into M – 1 buckets
  - Send all buckets to disk
- Step 2
  - Join every pair of corresponding buckets

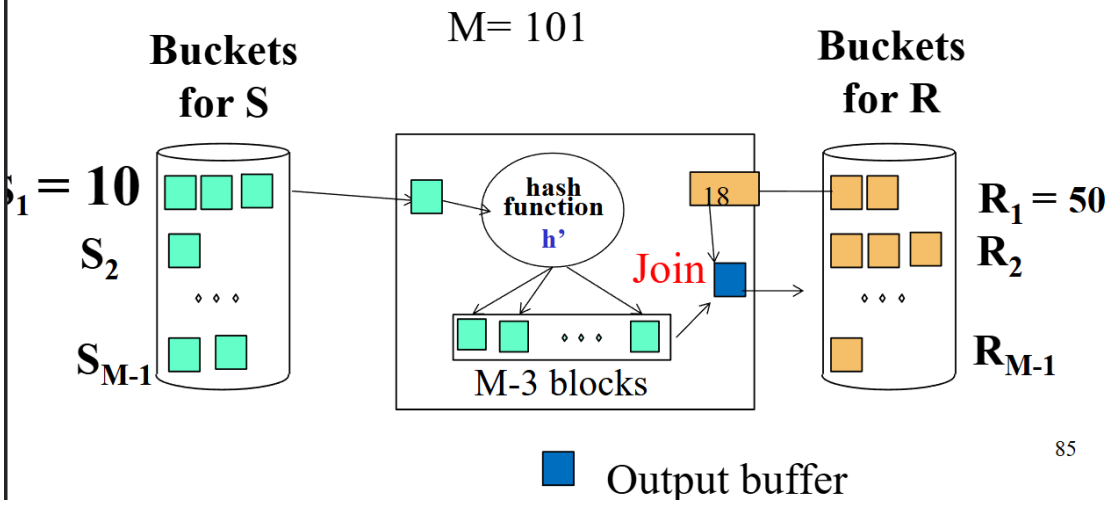
Partitioned Hash-Join

- Partition tuples in R and S using join attributes as key for hash
- Tuples in partition R_i only match tuples in partition S_i.
- R.age = S.age
- h(r.age) = h(25) = 2
- h(s.age) = h(25) = ?



Partitioned Hash-Join: Second Pass

- Read in a partition of S, say S_i, hash it using another hash function h'
- Load the matching partition R_i, one block at a time, output joining tuples.



Partitioned Hash Join

- Cost: 3B(R) + 3B(S)
- Assumption: min(B(R), B(S)) <= M^2
  - Or to be more exact: min(B(R), B(S))/(M-1) <= M-3
  - Or min(B(R), B(S))/(M-1) <= M-2 (if we do not use hash table to speed up the lookup)

Hashing notes

- Suppose M = 101 blocks (i.e., pages), B(R) = 1,000 blocks, B(S) = 5000 blocks (R.a = S.a)

- hash R into 100 buckets, 10 blocks/bucket: 2B(R)
  - R1, R2, …, R100
- hash S into 100 buckets, 50 blocks/bucket: 2B(S)
  - S1, S2, …, S100

- Join R1 with S1, R2 with S2, …, R100 with S100
  - cost: B(R) + B(S)

- total cost: 3B(R) + 3B(S)

what if

- M = 101, B(R) = 1,000, B(S) = 50, 000 blocks

- partition
  - R => R1, …, R100, 10 blocks/bucket, cost: 2B(R)
  - S => S1, …, S100, 500 blocks/bucket, cost: 2B(S)

R1: B(R)/(M-1)
S1: B(S)/(M-1)

- join
  - how to join R1 with S1

min(R1, S1)<=(M-2)

    - load R1 entirely into memory (10 blocks used)
    - load S1 90 blocks at a time, S11, S12, …, S16
  - cost: B(R) + B(S)

what if

- M = 101, B(R) = 20,000, B(S) = 50, 000 blocks
- partition
  - R => R1, …, R100, 200 blocks/bucket, cost: 2B(R)
  - S => S1, …, S100, 500 blocks/bucket, cost: 2B(S)
- join
  - how to join R1 (200) with S1 (500)
    - partition R1 => R11,…, R1,100 (2 blocks/bucket), cost: 2B(R)
    - partition S1 => S11,…, S1,100 (5 blocks/bucket), cost: 2B(S)
    - join R11 with S11, …, cost: B(R) + B(S)
  - cost: 5B(R) + 5B(S)

Hashing notes

- Suppose M = 101 blocks (i.e., pages), B(R) = 20,000 blocks, B(S) = 50,000 blocks (R.a = S.a)
- Step 1 (hash function = h)
  - hash R using h into 100 buckets, 200 blocks/bucket: 2B(R)
    - R1, R2, …, R100 (cost: 2B(R))
  - hash S using h into 100 buckets, 500 blocks/buck: 2B(S)
    - S1, S2, …, S100 (cost: 2B(S))
- Step 2 (hash function = h') question: h' =? h
  - Join R1 (200 blocks) with S1 (500 blocks)
    - step 1: partitioning (R1, R2, …, R100) cost: 2B(R)
    - step 2: partitioning (…

Sort-based vs. Hash-based Algorithms

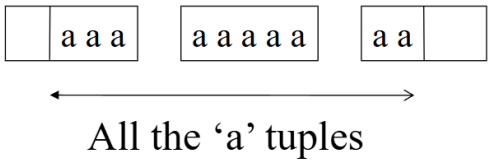
- Hash-based algorithms for binary operations have a size requirement only on the smaller of two input relations
- Sort-based algorithms sometimes allow us to produce a result in sorted order and take advantage of that sort later
- Hash-based algorithm depends on the buckets being of equal size, which may not be true if data are skewed

Index-Based Algorithms

- The existence of an index on one or more attributes of a relation makes available some algorithms that would not be feasible without the index
- Useful for selection operations
- Also, algorithms for join and other binary operations use indexes to good advantage

Clustered indexes

- In a clustered index, all tuples with the same value of the search key appear on roughly as the number of blocks as can hold them
    - That is, they are clustered together



Index Based Selection

- Selection on equality: $\sigma_{a=v}(R)$
- Clustered index on attribute a: cost = B(R)/V(R,a)
- Unclustered index on a: cost = T(R)/V(R,a)

We here ignore the cost of reading index blocks

Cost parameters

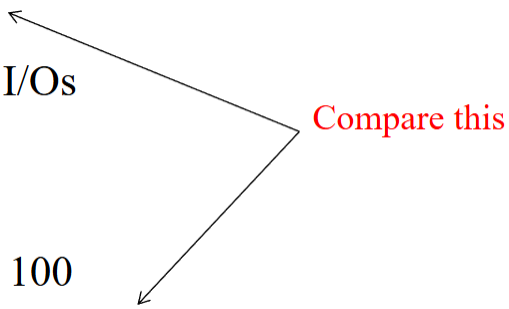
- **M** = number of blocks/pages that are available in main memory
- **B(R)** = number of blocks holding R
- **T(R)** = number of tuples in R
- **V(R,a)** = number of distinct values of the attribute a of R

Index Based Selection

- Example: B(R) = 2000, T(R) = 100,000, V(R, a) = 20, compute the cost of $\sigma_{a=v}(R)$

- Cost of using table scan:
  - If R is clustered: B(R) = 2000 I/Os
  - If R is unclustered: T(R) = 100,000 I/Os

- Cost of index-based selection:
  - If index is clustered: B(R)/V(R,a) = 100
  - If index is unclustered: T(R)/V(R,a) = 5000



Index-Based Join

- R ⋈ S
- Assume S has an index on the join attribute
- Iterate over R, for each tuple, fetch corresponding tuple(s) from S
- Assume R is clustered. Cost:
  - If index is clustered: B(R) + T(R)B(S)/V(S,a)
  - If index is unclustered: B(R) + T(R)T(S)/V(S,a)
- Compare this to NLJ (both R & S clustered)
  - B(R) + B(R)/(M-2) * B(S)

Indexed-Based Join vs. NLJ

- Index-based (R clustered, clustered index S.a)
  - B(R) + T(R)B(S)/V(S,a)
- NLJ (R & S clustered)
  - B(R) + B(R)/(M-2) * B(S)

- Index-Based wins if:
  - T(R)/V(S,a) < B(R)/(M-2), or
  - V(S,a) > (M-2) * T(R)/B(R)

※ **NLJ**: **Nested Loop Join**

Example

- Suppose M = 102 blocks (i.e., pages)
- R(a, b) ⋈ S(a, c)
- S has an index on attribute "a" and V(S,a) = 100
- B(R) = 1,000 blocks, B(S) = 5,000 blocks
- T(R) = 10,000 tuples, T(S) = 50,000 tuples

- Cost of R ⋈ S using index-based join algorithm
  - Index on S.a is clustered
  - Index on S.a is unclustered

Lec 12. Hadoop MapReduce

Hadoop

- A large-scale distributed batch-processing infrastructure
- Large-scale:
  - Handle a large amount of data and computation
- Distributed:
  - Distribute data & work across a number of machines
- Batch processing
  - Process a series of jobs without human intervention

Key components

- HDFS (Hadoop distributed file system)
  - Distributed data storage with high reliability
- MapReduce
  - A parallel, distributed computational paradigm
  - With a simplified programming model

HDFS

- Data are distributed among multiple data nodes
  - Data nodes may be added on demand for more storage space
- Data are replicated to cope with node failure
  - Typically replication factor = 2/3
- Requests can go to any replica
  - Removing the bottleneck (in single file server)

特征

- Data are distributed among multiple data nodes
  - Data nodes may be added on demand for more storage space
- Data are replicated to cope with node failure
  - Typically replication factor = 2/3
- Requests can go to any replica
  - Removing the bottleneck (in single file server)

HDFS has ...

- A single **NameNode**, storing meta data:
  - A hierarchy of directories and files
  - Attributes of directories and files
  - Mapping of files to blocks on data nodes
- A number of **DataNodes**:
  - Storing contents/blocks of files

HDFS also has ...

- A SecondaryNameNode
  - Maintaining checkpoints of NameNode
  - For recovery
- In a single-machine setup
  - all nodes correspond to the same machine

Block size

- HDFS: 128MB
  - Much larger than disk block size (4KB)
- Why larger size in HDFS?
  - Reduce metadata required per file
  - Fast streaming read of data (since larger amount of data are sequentially laid out on disk)
  - Good for workload with largely sequential read of large file

MapReduce

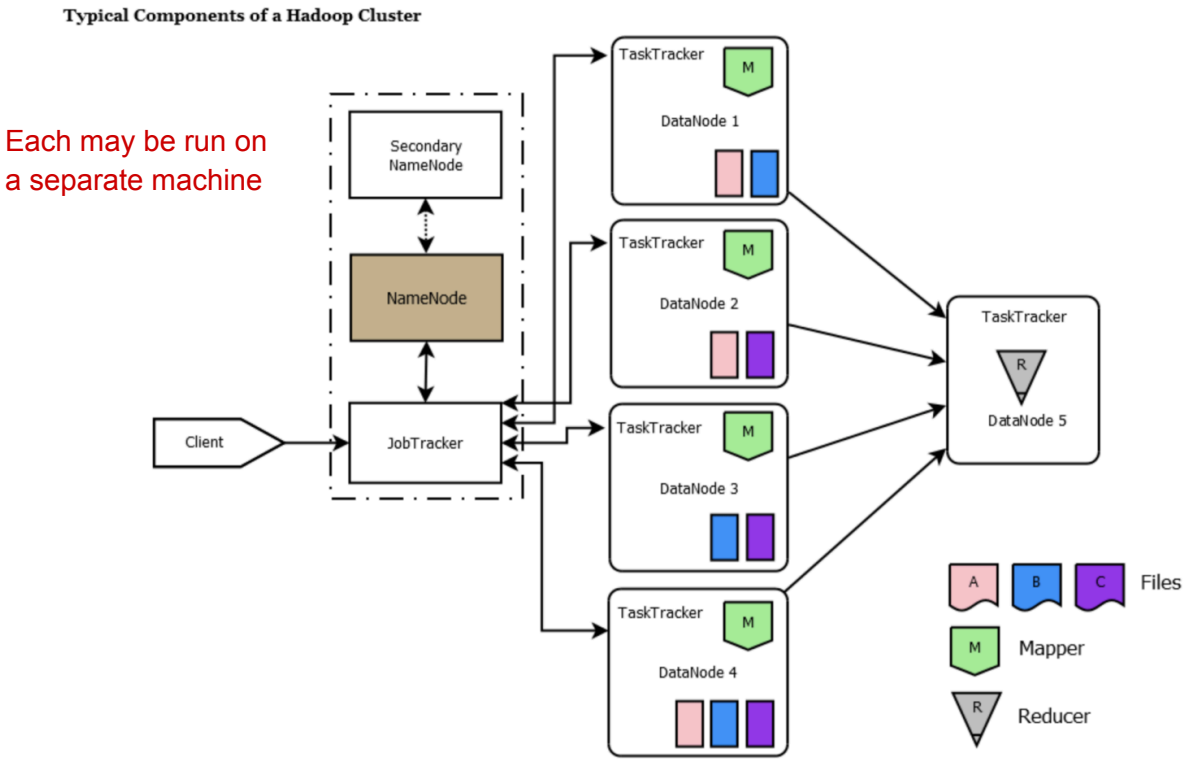
MapReduce job

- A MapReduce job consists of a number of
  - ***Map*** tasks
  - ***Reduce*** tasks
  - (Internally) ***shuffle*** tasks

***Map*, *reduce*, and *shuffle* tasks**

- *Map* task performs data transformation
- *Reduce* task combines results of map tasks
- *Shuffle* task sends output of map tasks to right reduce tasks
  - M1, M2 (=> key-values, output of Mapper)
    / \
  - R1 R2 (Reducer)

Hadoop cluster



Job tracker

- Takes requests from clients (MapReduce programs)
- Ask name node for location of data
- Assign tasks to *task trackers* near the data
    - Compared to: bring data to computation
- Reassign tasks if failed

Task tracker

- Accept (map, reduce, shuffle) tasks from job trackers
- Send heart beats to *job trackers*: I am alive
- Monitor status of tasks and notify job tracker

Lambda function

list = [1, 2, 3]

- list1 = map(lambda x: x ** 2, list)
- z = reduce(lambda x, y: x + y, list)

⇒

- z = reduce(f, list) where f is add function
  - Initially, z (an accumulator) is set to list[0]
  - Next, repeat z = add(z, list[i]) for each i > 0
  - Return final z

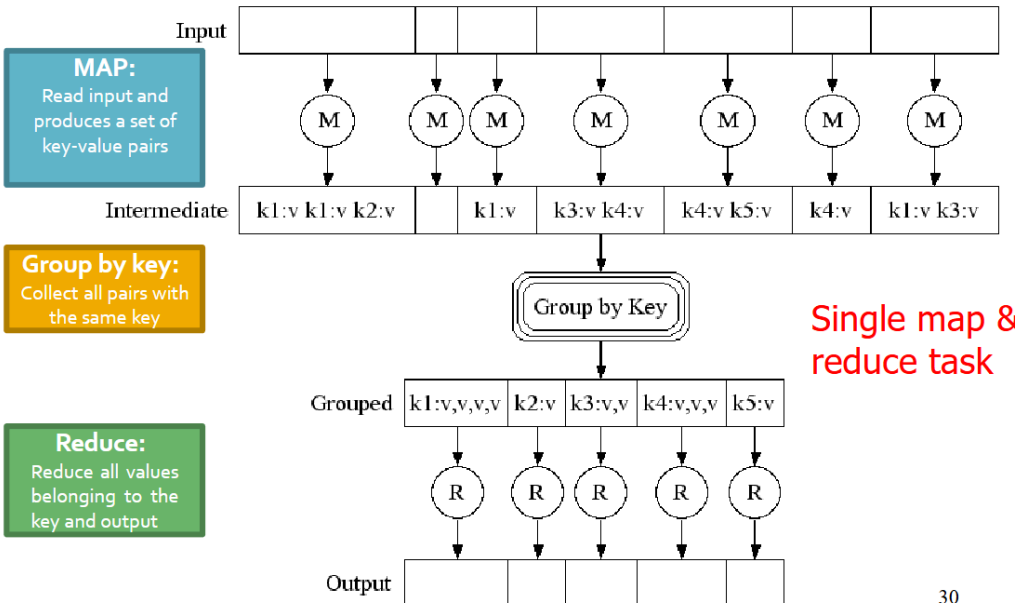
🌰 z = reduce(add, [1, 2, 3])

i = 0, z = 1; i = 1, z = 3; i = 2, z = 6

MapReduce

- Map function:
  - Input: <k, v> pair
  - Output: a list of <k', v'> pairs // ('LA', 2), ('LA', 3)
- Reduce function: ('LA', [2, 3])
  - Input: <k', list of v's> (note k's are output by map)
  - Output: a list of <k'', v''> pairs

Map-Reduce: A diagram



WordCount: Mapper

Object can be replaced with LongWritable

Data types of input key-value

Data types of output key-value

```
public class WordCount {

  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
```

Key-value pairs with specified data types

WordCount: Reducer

Data types of input key-value
Should be the same as output data types of mapper

Data types of output key-value

```
public static class IntSumReducer
     extends Reducer<Text,IntWritable,Text,IntWritable> {
  private IntWritable result = new IntWritable();

  public void reduce(Text key, Iterable<IntWritable> values,
                     Context context
                     ) throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val : values) {
      sum += val.get();
    }
    result.set(sum);
    context.write(key, result);
  }
}
```

A list of values

Mapper and Reducer

- Each map task runs an instance of **Mapper**
  - Mapper has a map function
  - Map task invokes the map function of the Mapper once for each input key-value pair

- Each reduce task runs an instance of **Reducer**
    - Reducer has a reduce function
    - Reduce task invokes the reduce function of the Reducer once for every different intermediate key

Shuffling

- Process of distributing intermediate key-values to the right *reduce tasks*

- It is **the ONLY communication** among *map* and *reduce tasks*
    - Individual map tasks do not exchange data directly with other map tasks
    - They are not even aware of existence of their peers

Shuffling

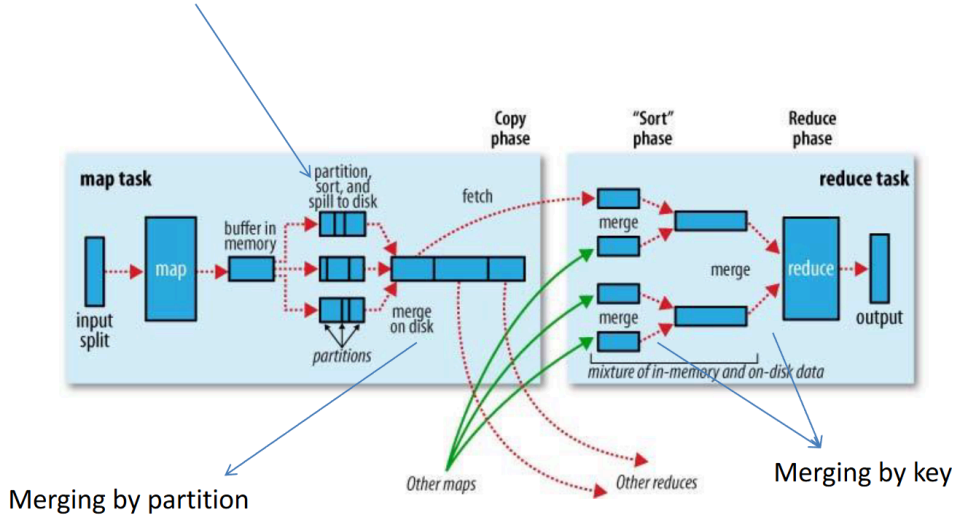
- Begins when a *map* task completed on a node
- All intermediate key-value pairs with the same key are sent to the same *reducer* task
- Partitioning method defined in Partitioner class
    - Default rule: partition by hashing the key

Internals of shuffling

- Map side
    - Partition, sort, spill & merge
- Reduce side
    - Fetch & merge

Shuffling process

Keys in the same partition are sorted (keys from different partitions may not be)



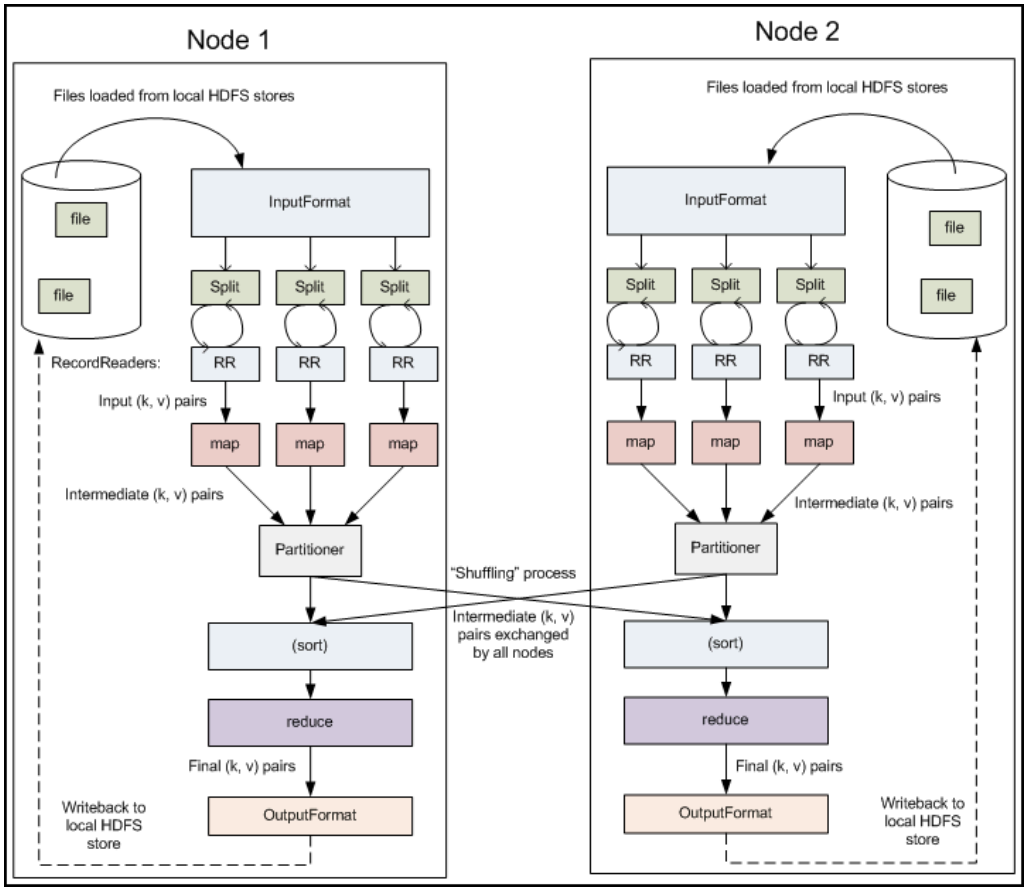
Map side

- Partition data in the buffer into *R* parts
  - *R* = # of reduce tasks
- Sort data in each partition by key
- Spill/write data in the buffer to disk
- Merge the spills
- Notify *job tracker*: output complete

Reduce side

- *Task tracke*r notified by *job tracker*: data ready
- Fetch/copy data from map side
- Merge the data
  - Some data may sit on disk once fetched
  - This depends on the buffer size
- Figure out groups from sorted data

Input and output format



InputFormat

- Determine how input files are split and read
- Defined in the Java **interface** InputFormat
- Job:
    - Split input file into chunks called InputSplits
    - Implement RecordReader to read data from splits

FileInputFormat

- Job:
    - Takes paths to files
    - Read all files in the paths
    - Divide each file into one or more InputSplits
- Subclasses:
    - TextInputFormat
    - KeyValueTextInputFormat
    - SequenceFileInputFormat

InputSplits

- f a file is big, multiple splits may be created
    - Typical split size = 128MB
- A map task is created for each split
    - i.e., a chunk of some input file

RecordReader (**RR**)

- InputFormat defines an instance of RR
    - E.g., TextInputFormat provides LineRecordReader
- LineRecordReader
    - Form a key-value pair for every line of file
    - Data type for key: LongWritable; value: Text
- Reader is repeatedly called
    - Until all data in the split are processed

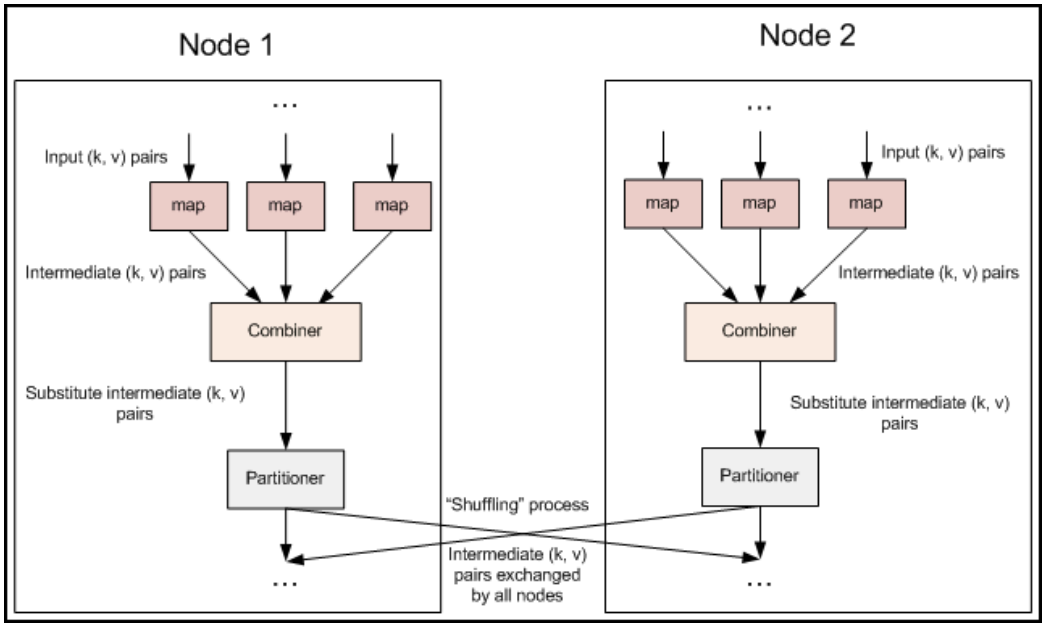
OutputFormat

- Define the format of output from Reducers
    - Output stored in a file
- Defined in the Java interface OutputFormat
- Implementation: FileOutputFormat
    - Subclasses: TextOutputFormat, SequenceFileOutputFormat

Outputs

- All Reducers write to the same directory
  - Each writes a separate file, named *part-r-nnnnn*
    - *r*: output from Reducers
    - *nnnnn*: partition id associated with reduce task
- Output directory
  - Set by FileOutputFormat.setOutputPath() method
- OutputFormat defines a RecordWriter
  - which handles the write

Combiner



Combiner

- Run on the node running the *Mapper*
  - Perform local (or mini-) reduction

- **Combine Mapper results**
  - Before they are sent to the *Reducers*
  - Reduce communication costs

- E.g., may use a combiner in WordCount
  - **(cat, 1), (cat, 1), (cat, 1) => (cat, 3)**
  - One key-value pair per unique word

Without combiner

- Mapper 1 outputs:
  - (cat, 1), (cat, 1), (cat, 1), (dog, 1)
- Mapper 2 outputs:
  - (dog, 1), (dog, 1), (cat, 1)
- Suppose there is only one Reducer
  - It will receive: **(cat, [1, 1, 1, 1]), (dog, [1, 1, 1])**

Implementing combiner

- May directly use the reduce function
  - If it is ***commutative*** and ***associative***
  - Meaning operations can be grouped & performed in any order

- Operation 'op' is *commutative* if:
  - A op B = B op A
- Op is *associative* if:
  - A op (B op C) = (A op B) op C

🌰 combiner

without combiner

- Consider two map tasks
  - M1 => 1, 2, 3 for some key x
  - M2 => 4, 5 for the same key
- Reducer adds all values for x
  - Result = (((1 + 2) + 3) + 4) + 5

with combiner

- M1 => 1, 2, 3 => combiner: (1 + 2) + 3 => 6
  M2 => 4, 5 => combiner: 4 + 5 => 9
- Reducer now **6 + 9**,
  - I.e., (((1 + 2) + 3) + (4 + 5))

※ **Question**:: is it the same as ((((1 + 2) + 3) + 4) + 5)?
Yes, since '+' is **associative**

- M1 => 1, 2, 3 => combiner: (1 + 2) + 3 => 6
  M2 => 4, 5 => combiner: 4 + 5 => 9
- Reducer may also compute **9 + 6**,
  - I.e., (4 + 5) + ((1 + 2) + 3)
  - Since values may arrive at reducer in any order

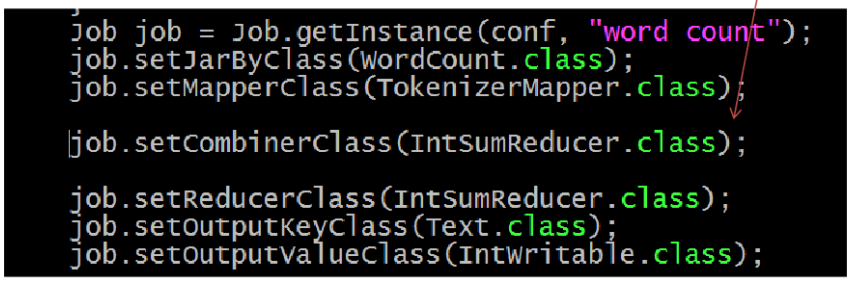
※ **Question**: is it the same as (((1 + 2) + 3) + 4) + 5?
Yes, since '+' is also **commutative**

Commutative and associative

- Examples
  – Sum
  – Max
  – Min
- Non-examples
  – Count
  – Average
  – Median

Enabling combiner

- job.setCombinerClass(IntSumReducer.class)
  – To use reduce function for combiner



MapReduce

Example

Id, name, age, gender

- Employee.txt

100,John,25,M
200,Mary,23,F
300,David,23,M
400,Bill,26,M
500,Jennifer,20,F
600,Maria,28,F

```
Map(key, value):
    Toks = tokenize(value, ',')
    Age = Toks[2]
    Gender = Toks[3]

    if (Age > 22):
      output(gender, 1)
```

```
SELECT gender, count(*)
FROM Emp
WHERE age > 22
GROUP BY gender
Having count(*) > 2
```

```
Reduce(key, values):
   cnt = sum(values) // expand it yourself
   if (cnt > 2):
     output(key, cnt)
```

M, 3
F, 2

Join

- R(A, B)⋈ S(A, C)
- Map:
  - r(a, b) => (a, ('R', b))
  - s(a, c) => (a, ('S', c))
- Reduce:
  - Joining every R tuple with every S tuple with same key
  - (a, [('R', b), ('S', c1), ('S', c2)]) => (a, (b, c1)), (a, (b, c2))

Lec 13. Nosql & Dynamo

CAP定理（CAP theorem)

- any distributed data store can provide only two of the following three guarantees

1. **Consistency** (Strong consistency)
   Every read receives the most recent write or an error. Note that consistency as defined in the CAP theorem is quite different from the consistency guaranteed in ACID database transactions.[4]
2. **Availability**
   Every request received by a non-failing node in the system must result in a response. This is the definition of availability in CAP theorem as defined by Gilbert and Lynch.[1] Note that availability as defined in CAP theorem is different from high availability in software architecture.[5]
3. **Partition tolerance**
   The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

Consequence

- A distributed system needs to tolerate *partitioning*
  - In other words, property *P* is required
  - Thus, when the network is partitioned, we need to choose between *availability* and (strong) *consistency*

    ⇒ viability of eventual consistency model

Eventual consistency model

- Acceptable to many applications
  - E.g., social media, cloud data storage, e-commerce

  🌰:
  - Amazon S3
  - Amazon DynamoDB (backbone of Amazon e-commerce and Web services)

NoSQL databases

- NoSQL: Not only SQL
- Key features
    - Flexible (non-relational) data model
    - Can be easily scaled out (horizontal scalability)
    - Data replicated over multiple servers
    - Weaker consistency model
    - High availability

Scale out vs. scale up

- Scale up (vertical scaling)
    - Beefing up a computer system
    - E.g., adding more CPUs, RAMs, and storage
- Scale out (horizontal scaling)
    - Adding more (commodity) computers
    - Moving some data to new computers

Types of NoSQL databases

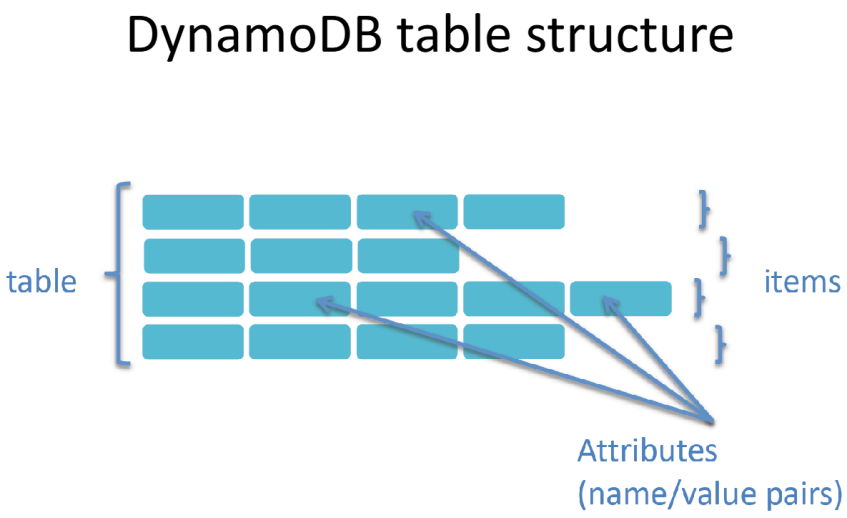
- Key-value stores
    - Redis
- Document stores
    - Firebase: entire database is a JSON value
    - MongoDB: database -> collections/tables -> JSON docs
    - DynamoDB: database -> tables -> rows -> key-value pairs
- Wide column stores
    - Database -> tables -> rows & columns
    - Different rows may have different columns
    - E.g., Apache Cassandra & HBase

Amazon DynamoDB

- Schema-less: no predefined schema
    - Other than primary key
- Database contains a list of tables, e.g., music
- A table consists of a set of items/rows
    - E.g., a set of music CDs
- Each **item** contains a set of attributes
    - E.g., artist, title, year of CD

Items

- Similar to rows in relational databases
- But different rows may have different set of attributes
- Max size of an item: `400K`
- No concept of columns in DynamoDB



Partition key

- Partition key
  - Partition (by hashing) the data across hosts for scalability & availability
- Pick an attribute with wide range of values & evenly distributed patterns for partition key
  - E.g., user ID
- E.g., artist name
  - Hash function may put "Rod Stewart" and "Maria Kelly" in the same partition

Sort key

- Allow searching within a partition
  - E.g., year
  - So primary key = artist + year

  → Possible multiple items with the same artist but different years

- This allows search all CDs by a specific artist and produced in certain years

Lec 14. Spark dataframe

Adding a new column => a new dataframe

df.withColumn('c', fc.lit(1))

- **lit** (literal的缩写) 是 Spark SQL functions 中的一个函数，用于在 Spark SQL 表达式中创建字面量/常量值。

```
# 假设有一个 DataFrame
# id | name
# 1 | Alice
# 2 | Bob

# 添加一个新的列 'c'，值都是 1
df = df.withColumn('c', fc.lit(1))

# 结果变成：
# id | name | c
# 1 | Alice | 1
# 2 | Bob | 1


# 所有行添加常量值1
df.withColumn('constant', fc.lit(1))

# 添加字符串常量
df.withColumn('status', fc.lit('active'))

# 添加布尔常量
df.withColumn('is_valid', fc.lit(True))
```

★**Note**：不能直接使用 Python 的字面量，必须用 `lit()` 包装，因为 Spark 需要知道这是一个 SQL 表达式中的常量值。

df.withColumn('c', fc.when(df.b > 2, 1).otherwise(-1)).show(

似于 SQL 的 *CASE WHEN*：

```
CASE
    WHEN b > 2 THEN 1
```

```
            ELSE -1
        END as c

# 假设原始数据：
# a | b
# 10 | 3
# 20 | 1
 # 30 | 4

# 执行上述代码后的结果：
# a | b | c
# 10 | 3 | 1 (因为 3 > 2)
# 20 | 1 | -1 (因为 1 不 > 2)
# 30 | 4 | 1 (因为 4 > 2)
```

Selection/filtering

1. `country[country.GNP > 10000]`
    - Pandas 风格的语法
    - 使用布尔索引进行过滤
    - 直观易读但在 Spark 中性能可能不是最优
2. `country.filter(country.GNP > 10000)`
    - 使用 DataFrame 列引用对象进行过滤
    - 可以利用 Spark 的优化器
    - 类型安全，IDE 可以提供更好的代码补全
3. `country.filter("GNP > 10000")`
    - 使用 SQL 风格的字符串条件
    - 简洁但不提供类型检查
    - 容易写出 SQL 风格的复杂条件
4. `country.filter('GNP > 10000 and GNP < 50000')`
    - SQL 风格的多条件过滤
    - 可以直接写复杂的逻辑表达式
    - 语法类似 SQL 的 WHERE 子句
5. `country.where('GNP > 10000')`
    - `where` 是 `filter` 的别名
    - 功能完全相同
    - 更接近 SQL 语法习惯

```
country[(country.GNP > 10000) & (country.GNP < 50000)]
```

- 使用小括号 ( ) 分组每个条件
- 使用 & 而不是 Python 的 and 运算符
- 这是因为 Spark 在处理 DataFrame 时需要构建表达式树

等价的其他写法：

```
# 使用 filter 方法
country.filter((country.GNP > 10000) & (country.GNP < 50000))

# 使用 SQL 字符串
country.filter("GNP > 10000 AND GNP < 50000")

# 使用 where 方法
country.where((country.GNP > 10000) & (country.GNP < 50000))
```

🌰 filter：

```
# 创建示例数据
data = [
    ("张三", 25, "北京", 5000),
    ("李四", 30, "上海", 8000),
    ("王五", 35, "广州", 10000),
    ("赵六", 28, "北京", 6000),
    ("孙七", 22, "上海", 4500)
]
df = spark.createDataFrame(data, ["name", "age", "city", "salary"])

# 1. 使用列名进行简单过滤
result1 = df.filter(col("age") > 25)

# 2. 使用 SQL 表达式
result2 = df.filter("city = '北京' and salary >= 5000")

⇒ 等价于： result2 = df.filter((df.city == "北京") & (df.salary >= 5000))

# 3. 多个条件组合（使用 &、|）
result3 = df.filter((col("age") >= 25) & (col("salary") > 5000))

# 4. 使用 isin() 进行多值匹配
cities = ["北京", "上海"]
result4 = df.filter(col("city").isin(cities))

# 5. 使用 like 进行模糊匹配
result5 = df.filter(col("name").like("张%"))
```

```
# 6. 使用 isNull() 和 isNotNull() 处理空值
result6 = df.filter(col("salary").isNotNull())

# 7. 使用 between 进行范围过滤
result7 = df.filter(col("age").between(25, 30))

# 8. 使用 startswith() 进行前缀匹配
result8 = df.filter(col("name").startswith("张"))
```

Groupby without aggregation

分组(groupBy)和获取唯一值(distinct)

分组操作：

```
# 这两种写法效果相同
country.groupBy('Continent')  # Spark 风格
country.groupby('Continent') # Pandas 风格

# 但只分组是不够的，需要聚合操作
country.groupBy('Continent').count() # 计算每个大洲的国家数量
country.groupBy('Continent').sum('Population') # 计算每个大洲的人口总和
country.groupBy('Continent').avg('GNP') # 计算每个大洲的平均 GNP
```

获取唯一值：

```
# 方法1：使用 distinct() 获取唯一的大洲
country[['Continent']].distinct()

# 方法2：使用 groupBy() 实现相同效果
country.groupBy('Continent').count()[['Continent']]
```

Aggregation w/o groupby

1. 使用字典方式：

```
country.agg({'GNP': 'max'})
```

- 使用字典，键是列名，值是聚合函数名
- 简单直观但功能有限

2. 使用 Spark SQL 函数：

```
import pyspark.sql.functions as fc
country.agg(fc.max('GNP').alias('max_gnp'))
```

- 更灵活和强大
- 可以使用更复杂的聚合函数
- alias() 用于给结果列命名

更多聚合操作示例：

```
# 多个聚合
country.agg(
    fc.max('GNP').alias('max_gnp'),
    fc.min('GNP').alias('min_gnp'),
    fc.avg('GNP').alias('avg_gnp'),
    fc.count('GNP').alias('count_gnp')
)

# 使用表达式
country.agg(
    fc.sum('Population').alias('total_population'),
    (fc.sum('GNP') / fc.sum('Population')).alias('avg_gnp_per_capita')
)

# 混合使用不同方式
country.agg(
    {'Population': 'sum'}, # 字典方式
    fc.max('GNP').alias('max_gnp') # 函数方式
)
```

- country.agg(fc.max('GNP').alias('max_gnp'), fc.min('GNP').alias('min_gnp')).show()

```
+---------+-------+
|  max_gnp|min_gnp|
+---------+-------+
|8510700.0|    0.0|
+---------+-------+
```

Group by with aggregation

Spark DataFrame API 写法：

```
country.groupBy('Continent').agg(
    fc.max("GNP").alias("max_gnp"),
    fc.count("*").alias("cnt")
).show()
```

等价的 SQL 写法：

```
SELECT
        Continent,
        max(GNP) as max_gnp,
        count(*) as cnt
FROM country
GROUP BY Continent
```

Group by with having

1. Spark DataFrame API 写法：

```
import pyspark.sql.functions as fc

country.groupBy('Continent').agg(
        fc.max("GNP").alias("max_gnp"),
        fc.count("*").alias("cnt")
        )\
        .filter('cnt > 5')\
        .show()
```

2. 等价的 SQL 写法：

```
SELECT
        Continent,
        max(GNP) as max_gnp,
        count(*) as cnt
FROM country
GROUP BY Continent
HAVING cnt > 5
```

关键对应关系：

- DataFrame 中的 `.filter()` 对应 SQL 中的 `HAVING`
- 这是因为是在聚合后进行的过滤，而不是在原始数据上过滤

```
# 同时使用 WHERE 和 HAVING 的效果
country.filter('GNP > 1000') # 相当于 WHERE
        .groupBy('Continent')
        .agg(
                fc.max("GNP").alias("max_gnp"),
                fc.count("*").alias("cnt")
        ) .filter('cnt > 5') # 相当于 HAVING
```

```
# SQL 等价写法
 """
SELECT Continent, max(GNP) as max_gnp, count(*) as cnt
FROM country
WHERE GNP > 1000
GROUP BY Continent
HAVING cnt > 5
"""
```

主要区别：

- WHERE (DataFrame 中的前置 filter): 在分组前过滤
- HAVING (DataFrame 中的后置 filter): 在分组后过滤
- DataFrame API 都使用 filter，但应用的时机不同
- SQL 明确区分 WHERE 和 HAVING

使用 **groupBy** 后接多个列，对每个分组计算汇总信息（如平均工资）。

```
grouped_df = df.groupBy("name", "department").avg("salary")

grouped_df.show()
```

Counting w/o group by

country.count()

- Note different from Pandas count()

=>

```
select count(*)
from country
```

Aggregating one or more columns

- country.groupBy('Continent').max('GNP').show()
- country.groupBy(['Continent', 'Region']).max('GNP').show()

Order by

1. 基本升序排序：

```
country.orderBy('Continent')
```

2. 降序排序（有三种写法）：

```
# 方式1：使用 fc.desc()
country.orderBy(fc.desc('Continent'))

# 方式2：使用列引用的 desc()
country.orderBy(country.Continent.desc())

# 方式3：使用字符串和 desc()
country.orderBy('Continent', ascending=False)
```

3. 多列排序：

```
# 所有列都是升序
country.orderBy("Continent", "GNP")

# 不同列不同顺序
country.orderBy(
        ['Continent', 'GNP'],
        ascending=[True, False] # Continent升序，GNP降序
)

# 使用 sort() 方法（orderBy 的别名） PREFERRED
country.sort(
        fc.desc('Continent'),
        fc.desc('GNP')
)

# 混合使用
 country.orderBy(
        'Continent',      # 默认升序
        fc.desc('GNP')          # 明确指定降序
)
```

1. DataFrame API 写法：

```
country
        # 先过滤 GNP
        .filter((country.GNP > 1000) & (country.GNP < 10000))
        # 按大洲和地区分组
        .groupBy('Continent', 'Region')
        # 计算平均寿命和计数
        .agg(
                fc.mean('LifeExpectancy').alias('avg_le'),
                fc.count('*').alias('cnt')
        )
        # 过滤分组结果
        .filter('cnt > 5')
        # 按平均寿命降序排序
        .orderBy(fc.desc('avg_le'))
        .show()
```

2. 等价的 SQL 写法：

```
SELECT
        Continent,
        Region,
        AVG(LifeExpectancy) as avg_le,
        COUNT(*) as cnt
FROM country
WHERE GNP > 1000 AND GNP < 10000
GROUP BY Continent, Region
HAVING cnt > 5
ORDER BY avg_le DESC
```

Join

1. 基本的等值连接：

```
country.join(city, country.Capital == city.ID)
```

- 相当于 SQL: `country JOIN city ON country.Capital = city.ID`
- 使用单个条件进行连接

2. 使用多个条件连接（方式1 - 使用 &）：

```
country.join(
        city,
        (country.Capital == city.ID) &
        (country.Population > city.Population)
)
```

- 使用 & 连接多个条件
- 需要用括号包裹每个条件

3. 使用多个条件连接（方式2 - 使用列表）：

```
country.join(
        City,
        [
                country.Capital == city.ID,
                country.Population > city.Population
        ]
)
```

- 使用列表包含多个条件
- 更简洁的语法

4. SQL 风格：(用 = 表示等值比较)

```
country.join(
   city,
   "Capital = ID AND Population > city.Population"
)
```

其他常见的 join 写法：

```
# 指定 join 类型

country.join(city, country.Capital == city.ID, "left") # LEFT JOIN

country.join(city, country.Capital == city.ID, "right") # RIGHT JOIN

country.join(city, country.Capital == city.ID, "outer") # FULL OUTER JOIN

country.join(city, country.Capital == city.ID, "inner") # INNER JOIN (默认)

# 使用表达式连接
country.join(
        city, fc.col("Capital") == fc.col("ID"),
)
```

两种写法完全相同：

```
# 方式1：位置参数
country.join(city, country.Capital == city.ID, "left")
# 方式2：命名参数
country.join(city, country.Capital == city.ID, how="left")
```

Natural join

- cl.join(city, 'CountryCode')
    - =: cl.join(city, cl.CountryCode == city.CountryCode)

```
SELECT *
FROM countrylanguage
NATURAL JOIN city
```

Union

获取数据子集：

```
# 获取美国的语言数据
usa = cl[(cl.CountryCode == 'USA')][['Language', 'IsOfficial']]

# 获取加拿大的语言数据
can = cl[(cl.CountryCode == 'CAN')][['Language', 'IsOfficial']]

# 直接通过 OR 条件获取两国数据
usa_can = cl[(cl.CountryCode == 'USA') | (cl.CountryCode == 'CAN')][['Language', 'IsOfficial']]
```

1. Bag Union (保留重复项):

```
# 两种写法等价
usa.union(can) # 推荐写法
usa.unionAll(can) # 旧写法
```

- 合并两个 DataFrame
- 保留所有重复的行
- 类似 SQL 的 UNION ALL

2. Set Union (去除重复项):

```
usa.union(can).distinct()
```

- 合并后去除重复行
- 类似 SQL 的 UNION

注意事项:

- union 要求两个 DataFrame 的列数和类型必须相同
- 常用于合并来自不同源但结构相同的数据
- 如果需要保留来源信息，最好先添加标识列

```
# 添加来源标识后合并
usa_with_source = usa.withColumn('source', fc.lit('USA'))
can_with_source = can.withColumn('source', fc.lit('CAN'))
usa_with_source.union(can_with_source)
```

多个表的 FROM

 Spark DataFrame 中，处理多个表的 FROM 有几种主要写法

```
from pyspark.sql.functions import col

# 创建示例数据
# 订单表
orders_data = [
    (1, "2024-01-01", 100, "user1"),
    (2, "2024-01-02", 200, "user2"),
    (3, "2024-01-03", 300, "user1")
]
orders = spark.createDataFrame(orders_data, ["order_id", "date",
"amount", "user_id"])
```

```
# 用户表
users_data = [
    ("user1", "John", "NY"),
    ("user2", "Mary", "CA")
]
users = spark.createDataFrame(users_data, ["user_id", "name",
"state"])

# 支付表
payments_data = [
    (1, "completed"),
    (2, "pending"),
    (3, "completed")
]
payments = spark.createDataFrame(payments_data, ["order_id",
"status"])



# 1. 使用多个join
result1 = orders.join(users, "user_id") \
               .join(payments, "order_id") \
               .select("order_id", "name", "amount", "status")


# 2. 使用SQL方式
orders.createOrReplaceTempView("orders")
users.createOrReplaceTempView("users")
payments.createOrReplaceTempView("payments")

result2 = spark.sql("""
    SELECT o.order_id, u.name, o.amount, p.status
    FROM orders o
    JOIN users u ON o.user_id = u.user_id
    JOIN payments p ON o.order_id = p.order_id
""")


# 3. 使用别名的DataFrame方式
o = orders.alias("o")
u = users.alias("u")
p = payments.alias("p")

result3 = o.join(u, o.user_id == u.user_id) \
          .join(p, o.order_id == p.order_id) \
          .select(o.order_id, u.name, o.amount, p.status)
```

```
# 4. 对于复杂条件的join
result4 = orders.join(
    users,
    (orders.user_id == users.user_id) & (users.state == "NY"),
    "inner"
).join(
    payments,
    orders.order_id == payments.order_id,
    "left"
).select("order_id", "name", "amount", "status")
```

Lec 14. Spark-RDD

Characteristics of Hadoop

- Acyclic data flow model
  - Data loaded from stable storage (e.g., HDFS)
  - Processed through a sequence of steps
  - Results written to disk
- Batch processing
  - No interactions permitted during processing

Problems of Hadoop

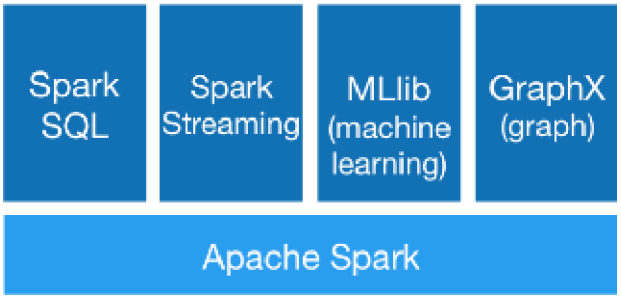
- ill-suited (不适合) for iterative algorithms that requires repeated reuse of data
  - E.g., machine learning and data mining algorithms such as k-means (clustering), PageRank, logistic regression
- ill-suited for interactive exploration of data
  - E.g., OLAP on big data

Spark

- Support working sets (of data) through RDD
  - Enabling reuse & fault-tolerance
- 10x faster than Hadoop in iterative jobs
- Interactively explore 39GB (Wikipedia dump) with sub-second response time
  - Data were distributed over 15 EC2 instances

Spark

- Provides libraries to support
  - embedded use of SQL
  - stream data processing
  - machine learning algorithms
  - processing of graph data



RDD: Resilient Distributed Dataset 弹性分布式数据集

- RDD
  - Read-only, partitioned collection of records
  - Operations performed on partitions in parallel
  - Maintain lineage for efficient fault-tolerance

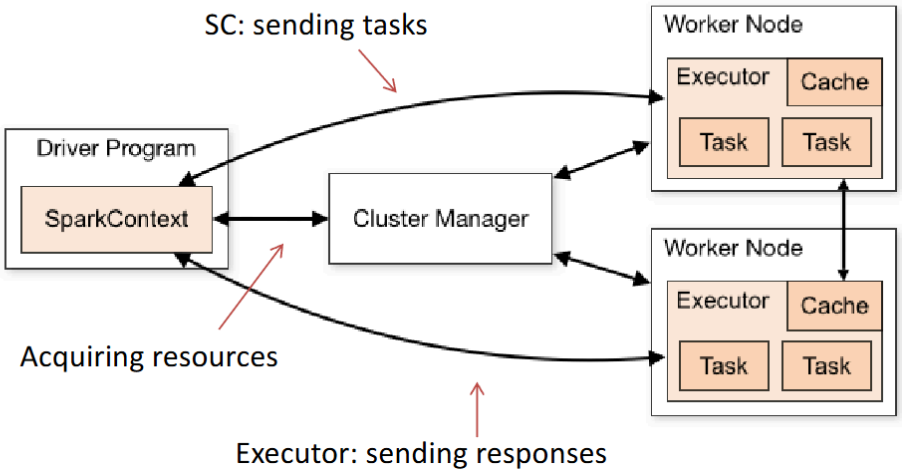
- Methods of creating an RDD
  - from an existing collection (e.g., Python list/tuple)
  - from an external file

- Distributed
  - Data are divided into a number of partitions
  - & distributed across nodes of a cluster to be processed in parallel

- Resilient
  - Spark keeps track of transformations to dataset
  - Enable efficient recovery on failure (no need to replicate large amount of data across network)

Architecture

- SparkContext (SC) object coordinates the execution of application in multiple nodes
  - Similar to Job Tracker in Hadoop MapReduce

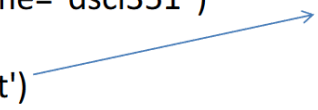


Components

- Cluster manager
  - Allocate resources across applications
  - Can run Spark's own cluster manager or
  - Apache YARN (Yet Another Resource Negotiator)
- Executors
  - Run tasks & store data

wc.py

```
from pyspark import SparkContext
from operator import add

sc = SparkContext(appName="dsci351")

lines = sc.textFile('hello.txt')

counts = lines.flatMap(lambda x: x.split(' ')) \
          .map(lambda x: (x, 1)) \
          .reduceByKey(add)

output = counts.collect()

for v in output:
   print(v[0], v[1])
```



SQL / Spark / RDD

假设我们要找出：每个大洲人口超过 100 万的城市数量，同时这些城市必须是所在国家的首都，并且国家 GNP 大于 5000。

1. SQL 写法：

```
SELECT
        co.Continent,
        AVG(ci.Population) as avg_city_pop,
        COUNT(*) as capital_count
FROM country co
JOIN city ci ON co.Capital = ci.ID
WHERE ci.Population > 1000000
        AND co.GNP > 5000
GROUP BY co.Continent
HAVING capital_count > 2
ORDER BY avg_city_pop DESC
```

2. Spark DataFrame API 写法：

```
from pyspark.sql import functions as fc

country.join(
        city,
        country.Capital == city.ID
).filter(
        (city.Population > 1000000) &
        (country.GNP > 5000)
).groupBy(
        'Continent'
).agg(
        fc.avg('Population').alias('avg_city_pop'),
        fc.count('*').alias('capital_count')
).filter( 'capital_count > 2'
).orderBy(
        fc.desc('avg_city_pop')
).show()
```

3. Spark RDD 写法：

```
# 假设我们已经有了 country_rdd 和 city_rdd
def parse_country(line):
        # 假设每行数据格式为: country_id,continent,capital,gnp
        fields = line.split(',')
        return (fields[0], (fields[1], int(fields[2]), float(fields[3])))

def parse_city(line):
        # 假设每行数据格式为: city_id,population
        fields = line.split(',')
        return (int(fields[0]), int(fields[1]))

result = country_rdd.map(parse_country) \
        .filter(lambda x: x[1][2] > 5000) \
        .map(lambda x: (x[1][1], (x[1][0], x[0]))) \
        .join(city_rdd.map(parse_city)) \
        .filter(lambda x: x[1][1] > 1000000) \
        .map(lambda x: (x[1][0][0], (x[1][1], 1))) \
        .reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1])) \
        .map(lambda x: (x[0], x[1][0]/x[1][1], x[1][1])) \
        .filter(lambda x: x[2] > 2) \
        .sortBy(lambda x: -x[1])

for continent, avg_pop, count in result.collect():
        print(f"{continent}: avg pop = {avg_pop}, count = {count}")
```