Advanced Databases

2017-2018

# 2.a.

## Is\_Member

Bytes per Tuple = 4 bytes (country) + 12 bytes (organisation) + 30 bytes (type) = 46 bytes

Tuples per Page = = 89 tuples per page

Number of Pages =

## Organisation

Bytes per Tuple = 12 bytes (abbreviation) + 35 bytes (city) + 4 bytes (country) + 3 bytes (established)

Tuples per Page = = 75 tuples per page

Number of Pages =

## Total = Is\_Member + Organisation = 899 pages + 67 pages = 966 pages

# 2.b.

## Outer Input

Number of Tuples = 0.75 x 80,000 = 60,000 tuples

Tuples per Page= 89

Number of Pages =

675 pages > 50 pages, so the outer input doesn’t fit in cache

Number of Page Faults = 675 page faults

## Inner Input

Number of Tuples = 5,000 tuples

Tuples per Page = 75

Number of Pages =

67 pages > 50 pages, so inner input doesn’t fit in cache

Number of Page Faults = 675 (pages in outer relation) x 67 = 45,225 page faults

## Total Page Faults = 675 (outer) + 45,225 (inner) = 45,900 page faults

# 2.c.

5000 tuples (abbreviation column = 12B) => 5000 \* 16 = 80000B => 15 pages (fit in memory buffer)

Is\_member (80000\*0.75 = 60000) => 60000 \* |12 + 4 = 720000B => 180pages

15 \* 235 = 3525 page faults, since the inner input fits in cache we only take into account the outer input.

# 2.d.

|  |  |
| --- | --- |
| Slot | Value |
| 0 | 0, 0, 0 |
| 1 | 9, 9 |
| 2 | 17, 17, 17 |
| 3 | 3 |
| 4 | 12, 12, 12 |
| 5 | 21 |
| 6 | 11 |
| 7 | 8, 8 |

**Alternative** (based on the algorithm in the slides, value is the attribute we are aggregating, then the actual aggregate, which is assumed to be a count, is stored in the aggregate column)

|  |  |  |
| --- | --- | --- |
| Slot, i.e f(x) | Value | Aggregate |
| 0 | 0 | 3 |
| 1 | 9 | 2 |
| 2 | 17 | 3 |
| 3 | 3 | 1 |
| 4 | 12 | 3 |
| 5 | 21 | 1 |
| 6 | 11 | 1 |
| 7 | 8 | 2 |

# 2.e.

Total = 36 probes

(Note: a probe is every time you look at a slot - even if there isn’t a conflict)

# 2.f.

## Function Call Cost

325,195 function calls x 15 cycles per call = 4,877,925 cycles

## Join Hash-Table

40 pages < 50 pages (size of the buffer pool).

Therefore, the Join hash table fits in the cache and there is no I/O.

## Aggregation Hash Table

Fits in the cache, no I/O.

## Overall: the query is CPU-bound, as the cost of function calls exceeds the cost of I/O

Hash tables fit in memory so no I/O

Total I/O = is\_member scan costs + organisation scan costs = (67 + 899) \* 6000 = 5796000 cycles

Therefore, I/O bound

3a. Pretty sure this is wrong but here goes…

organizations = FOREACH organization GENERATE abbreviation, city;

countries = FOREACH country GENERATE name, code, population;

members = FILTER is\_member BY type='member';

member\_countries = JOIN members BY country, countries BY code;

organization\_group = GROUP member\_countries BY organization;

----------------------------------------------------------------------------------------------------------------------------  
THEN THIS

top\_four = FOREACH organization\_group {

top = TOP(4, population, organization\_group)

ordered = ORDER top BY population;

GENERATE organization, ordered

}

organization\_join = JOIN top\_four BY organization,

organizations BY abbreviation;

ordered = ORDER organization\_join BY abbreviation;

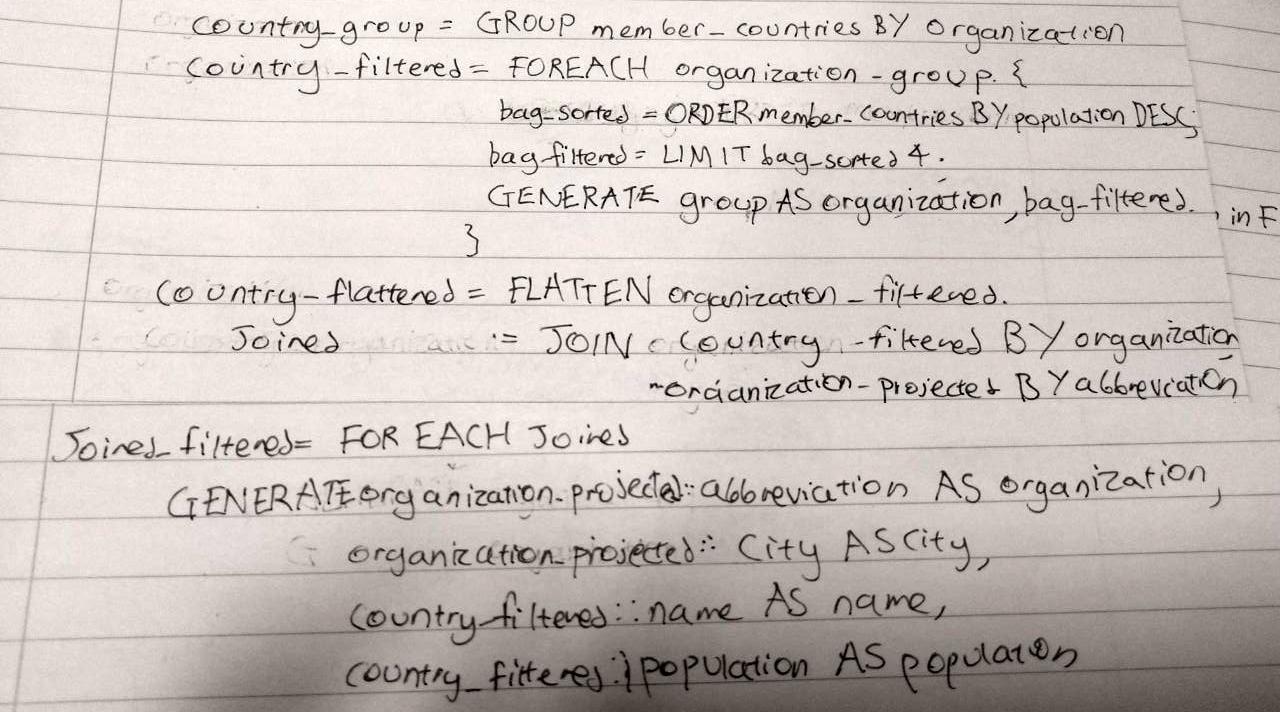
flattened = FLATTEN ordered;

result = FOREACH ordered

GENERATE organization, city, group::co

untry.name, group::country.population;

--------------------------------------------------------------------------------------------------------------------------

OR  


--------------------------------------------------------------------------------------------------------------------------

### 3b

The member\_countries join can be spread over S1 and S2 since only countries with code <= L are on S1 and all member data for these countries is also on S1. Then you can do the equivalent on S2 for all countries with code >L. This means the joins are smaller and more efficient.

Because the keys are sorted, we could add a ‘merge’ keyword after the JOIN

### 3c

**LTM11:**

SELECT is\_member.organiszation

country.code

country.population

INTO ltm11

FROM country

JOIN is\_member

ON is\_member\_country = country.code

WHERE type = ‘member’

**LTM21:**

SELECT is\_member.organiszation

country.code

country.population

INTO ltm21

FROM country

JOIN is\_member

ON is\_member\_country = country.code

WHERE type = ‘member’

**LTM12:**

SELECT organization.abbreviation

organization.city

INTO ltm12

FROM organization

**LTM22:**

SELECT organization.abbreviation

organization.city

INTO ltm22

FROM organization

**GTM:**

SELECT organization.abbreviation

organization.city

COUNT(member\_orgs.code) AS no\_member

AVG(member\_orgs.population) AS avg.pop

FROM (ltm11 UNION ltm21) AS member\_orgs

JOIN (ltm12 UNION ltm22) AS organization

ON member\_orgs.organization = organization.abbreviation

GROUP BY organization.abbreviation.

organization.city

### 3d

bordersi = borders SEMI countryi*(join on country1=code)*

encompassesi = encompasses SEMI countryi*(join on country=code)*

locatedi = located SEMI countryi*(join on country=code)*

### 4a

i)

**Local Join at S1:**

S2→ S1 = RowSize(Πname, population,codecountry) \* |country|

* RowSize(Πorganization, countryis\_member) \* |is\_member|

= (32+4+4)\*195 + (12+4)\*80000

= 1,287,800B

**Remote Join at S2:**

S2→ S1 = RowSize(Πcountry.name, country.population, is\_member.organizationcountry JOIN is\_member) \* |country JOIN is\_member|

= (32+4+12)\*80000

= 3,840,000B

Local Join is, therefore, more efficient

ii)

S1→ S2 = RowSize(σestablished > ‘1992-01-01’Πabbreviationorganization) \* |σestablished > ‘1992-01-01’organization|

= (12)\*(5000\*0.1)

= 6000B

S2→ S1 = RowSize(Πcountry.name, country.population, is\_member.organization organization SEMI country JOIN is\_member) \* |organization SEMI country JOIN is\_member|

= (32+4+12)\*(0.1)\*80000

= 384,000B

Total sent is 390,000B

## 4b

i)

Conflicts:

r1[CA] → w2[CA]

w2[CR] → r1[CR]

There is a cycle in the graph, and so it is NOT CSR

It is NOT RC, since w2[CR] → r1[CR] is a dirty read, and c1 occurs before c2, and so, therefore, it is not ACA or ST either.

The anomaly is a dirty read as stated above of CR, and H1 suffers from it

Also inconsistent analysis, and H\_1 suffers from it.

ii)

Lol idk looks hard

Maybe like 3 read and 3 write locks using predicate locking idk.

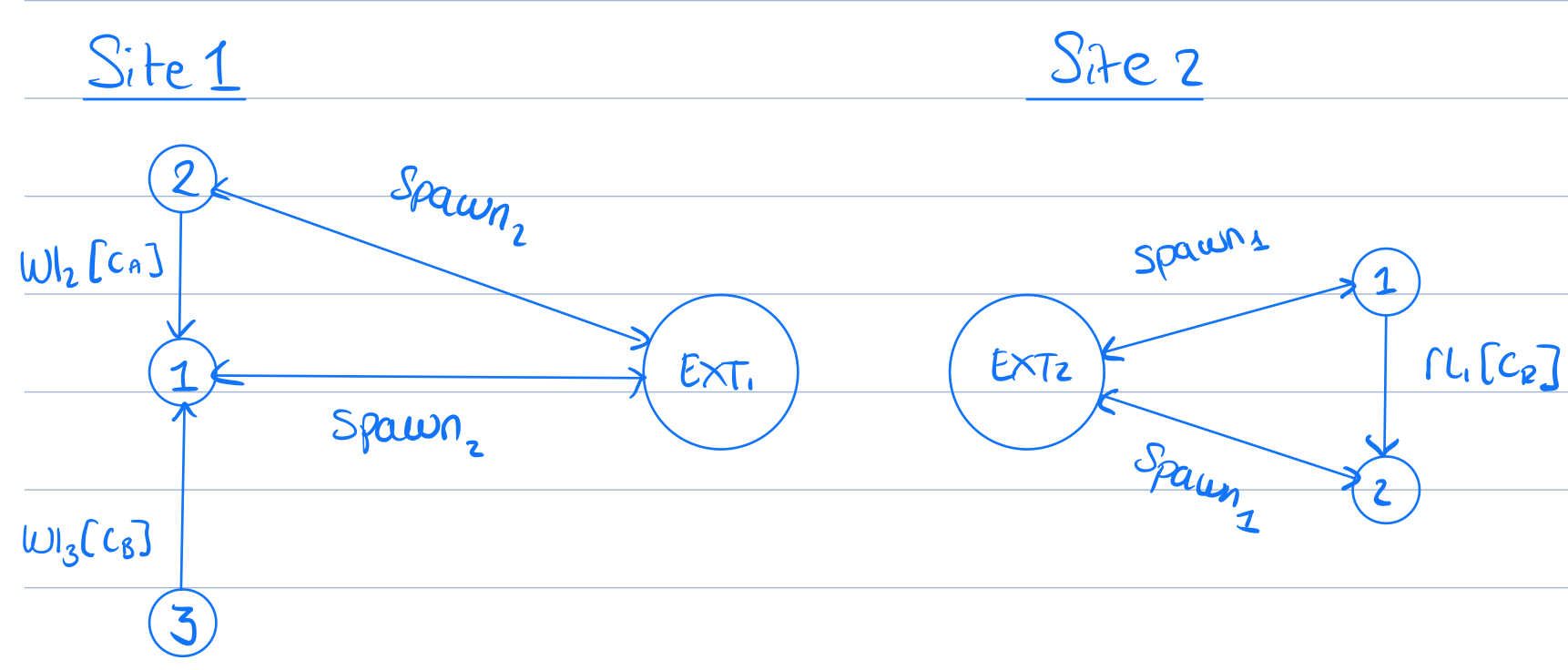
^I back that.

Maybe this is asking about justifications around using aggressive vs conservative 2PL. I think conservative should be used as aggressive will lead to a deadlock because of the conflicts listed above.

Copied from exercises answers:

Clearly the number of read operations is greater than the number of write operations, and a write-locks-all strategy would be optimal. Thus you would obtain write locks on all five sites, and read locks on one site.

iii)



### 4c

i)

|  |
| --- |
| **Code** |
| CH |
| CZ |
| GB |
| R |
| SK |

Code of all countries continuously present since SK existed

ii)

|  |
| --- |
| **Code** |
| CS |

All countries which only have a decreasing population over time (I’m v unsure of this one)

Alternative answer:

|  |
| --- |
| **Code** |
| CH |
| CS |
| GB |
| YU |
| SK |

Countries whose populations have never decreased. (+1)

+1

+27