Problems for Chapter 10: The Basics of Query Processing

1. What kind of indexing is usually needed to efficiently evaluate the following query?

```
SELECT E.Id
FROM Employee E
WHERE E.salary <= 100000 AND E.salary >= 30000
```

Solution: Secondary B+ tree index with search key `salary`

2. Consider a relation **s** over the attributes A and B with the following characteristics:

   - 7,000 tuples with 70 tuples per page
   - A hash index on attribute *A*
   - The values that the attribute *A* takes in relation **s** are integers that are uniformly distributed in the range 1 – 200.

   (a) Assuming that the aforesaid index on *A* is *unclustered,* estimate the number of disk accesses needed to compute the query $\sigma_{A=18}(\mathbf{s})$.

   (b) What would be the cost estimate if the index were clustered?

   Explain your reasoning.

   Solution:

   It takes 1.2 I/Os to find the right bucket. The number of tuples selected from **s** is expected to be 7000/200 = 35.

   (a) If the index is unclustered, every tuple can potentially lead to a separate I/O. So, the cost will be ceiling(1.2 + 35) = 37.

   (b) If the index is clustered, then 35 tuples can span at most 2 pages (2 rather than 1 because they might cluster around a page boundary), so the I/O cost would be ceiling(1.2+2)=4.

3. What is the cost of external sorting?

   Solution:

   $2 * F * log_{M-1}F$, where F is the number of pages in file and M is the number of pages in memory.

4. Give an example of an instance of the TRANSCRIPT relation (with the attributes StudId, CrsCode, Semester, and Grade) and a hash function on the attribute sequence ⟨StudId, Grade⟩ that sends two identical tuples in $\pi_{\texttt{StudId,Semester}}(\text{TRANSCRIPT})$ into *different* hash buckets. (This shows that such a hash-based access path cannot be used to compute the projection.)

   Solution: Consider the following two tuples: ⟨111111111, EE101, F1997, A⟩ and ⟨111111111,MAT123,F1997,B⟩. Since these tuples have identical projections on the attributes StudId,Semester, they become one in the projected relation

   $$\pi_{\texttt{StudId,Semester}}(\text{TRANSCRIPT})$$

   Consider now the following hash function on StudId,Grade:

   $$f(t) \;=\; (t.\texttt{StudId} + (\mathit{ascii}(t.\texttt{Grade}) - \mathit{ascii}(\texttt{'A'}))) \bmod 1000$$

   The first tuple will then be sent to the bucket 111111111 mod 1000 = 111 and the second to 111111112 mod 1000 = 112.

5. Estimate the cost of **r** $\bowtie$ **s** using

   (a) Sort-merge join

   (b) Block nested loops

   where **r** has 1,000 tuples, 20 tuples per page; **s** has 2,000 tuples, 4 tuples per page; and the main memory buffer for this operation is 22 pages long.

   Solution:

   Relation **r** has 50 pages and **s** has 500 pages. In each case we ignore the cost of outputting the result, as it is the same for both methods.

   (a) Sorting **r** will take 2*50*$log_{21}50 \approx 200$. Sorting **s** will take 2*500*$log_{21}500 \approx 3000$. Thus, assuming that merging can be done during the last phase of sorting, the total cost should not exceed 3200 page transfers.

   (b) Assuming that **r** is scanned in the outer loop, the cost is 50 + (50/20)*500 = 1300.

6. Consider the expression

$$\sigma_{\texttt{StudId=666666666} \wedge \texttt{Semester='F1995'} \wedge \texttt{Grade='A'}}(\textsc{Transcript})$$

Suppose that there are the following access paths:

(a) An unclustered hash index on `StudId`

(b) An unclustered hash index on `Semester`

(c) An unclustered hash index on `Grade`

Which of these access paths has the best selectivity and which has the worst? Compare the selectivity of the worst access path (among the above three) to the selectivity of the file scan.

Solution:

With the unclustered hash index on `StudId`, we will find exactly the bucket that contains all the transcript records for student with the Id 666666666. Since the index is unclustered, this access method will fetch (in the worst case) as many pages as the number of transcript records for that student. In our sample relation in Figure 4.5, this would be 3 pages. In a typical university, an undergraduate student would have to earn 120-150 credits. With 3 credits per course it would make 40-50 transcript records and, thus, the selectivity would be this many pages of data.

With the unclustered hash index on `Semester`, we jump to the bucket for the transcript records in the `F1995` semester and then we need to fetch all these records from the disk to check the other conditions. In a university with enrollment 20,000, selectivity of this access path can be as high as that. In our sample database, however, there are only two transcript records for Fall 1995.

With the unclustered hash index on `Grade`, we get into the bucket of the transcript records where the student received the grade `A`. If only 10% of the students get an `A`, the bucket woud hold 2,000 records per semester. In 20 years (2 semesters a year), the university might accumulate as many as 80,000 transcript records in that bucket. In our sample database, we have 5 transcript records where the student got an `A`.

Thus, in a typical university, the third access path has the worst selectivity and the first has the best. In the sample database of Figure 4.5, the second method has the best selectivity and the third the worst.

7. Compute the cost of $\mathbf{r} \bowtie_{\mathtt{A}=\mathtt{B}} \mathbf{s}$ using the following methods:

   (a) Nested loops

   (b) Block-nested loops

   (c) Index-nested loops with a hash index on B in **s**. (Do the computation for both clustered and unclustered index.)

   where **r** occupies 2,000 pages, 20 tuples per page, **s** occupies 5,000 pages, 5 tuples per page, and the amount of main memory available for block-nested loops join is 402 pages. Assume that at most 5 tuples in **s** match each tuple in **r**.

   Solution:

   (a) Nested loops: scan **r** and for each of its 40,000 tuples scan **s** once. The result is

   $$2,000 + 40,000 \times 5,000 = 200,002,000 \text{ pages}$$

   (b) Block-nested loops: Scan **s** once per each 400-page block of **r**, i.e., 5 times. The result therefore is:

   $$2,000 + 5,000\lceil \frac{2,000}{402-2} \rceil = 27,000 \text{ pages}$$

   (c) Index-nested loops: The result depends on whether the index on `B` in **s** is clustered or not. For the clustered case, all tuples of **s** that match a tuple of **r** are in the same disk block and require 1 page transfer (since we assumed that at most 5 tuples of **s** match, they all fit in one disk block). We also need to search the index once per each tuple of **r**. Suppose the later takes 1 disk access. Thus, the total is

   $$2,000 + (1 * 1 + 1.2) \times 40,000 = 90,000 \text{ pages}$$

   In case of an unclustered index, the matching tuples of **s** can be in different blocks. As before, assume that **s** has at most 5 matching tuples per tuple in **r**. Thus, the cost would be

   $$2,000 + (1 * 5 + 1.2) \times 40,000 = 250,000 \text{ pages}$$

Problems for Chapter 11: An Overview of Query Optimization

1. Suppose a database has the following schema:
   TRIP(fromAddrId: INTEGER, toAddrId: INTEGER, date: DATE)
   ADDRESS(id: INTEGER, street: STRING, townState: STRING)

   (a) Write an SQL query that returns the street of all addresses in 'Stony Brook NY' that are destination of a trip on '5/14/02'.

   (b) Translate the SQL query in (a) into the corresponding "naive" relational algebra expression.

   (c) Translate the relational algebra expression in (b) into an equivalent expression using pushing of selections and projections.

   (d) Translate the relational algebra expression in (c) into a most directly corresponding SQL query.

   Solution:

   (a)
   ```
   SELECT A.street
   FROM ADDRESS A, TRIP T
   WHERE A.id=T.toAddrId AND A.townState='Stony Brook NY'
         AND T.date='05/14/02'
   ```

   (b) $\pi_{\texttt{street}}\sigma_{\texttt{id=toAddrId AND townState='StonyBrookNY' AND date='05/14/02'}}(\texttt{Address} \times \text{TRIP})$

   (c) $\pi_{\texttt{street}}(\sigma_{\texttt{townState='StonyBrookNY'}}(\text{ADDRESS}) \bowtie_{\texttt{id=toAddrId}} \sigma_{\texttt{date='05/14/02'}}(\text{TRIP}))$

   (d)
   ```
   SELECT A.street
   FROM (SELECT * FROM ADDRESS WHERE townState='Stony Brook NY') A,
        (SELECT * FROM TRIP WHERE date='05/14/02') T
   WHERE A.id=T.toAddrId
   ```

2. Consider a relation **r** over the attributes A, B, C with the following characteristics:

   - 5,000 tuples with 5 tuples per page
   - Attribute A is a candidate key
   - Unclustered hash index on attribute A
   - Clustered B^+ tree index on attribute *B*
   - Attribute *B* has 1,000 distinct values in **r**
   - Attribute *C* has 500 distinct values and an unclustered 3-level B^+ tree index

   (a) Estimate the cost of computing $\sigma_{A=const}(\mathbf{r})$ using the index

   (b) Estimate the cost of computing $\sigma_{B=const}(\mathbf{r})$ using the index

   (c) Estimate the cost of computing $\sigma_{C=const}(\mathbf{r})$ using the index

   (d) Estimate the cost of computing $\pi_{AC}(\mathbf{r})$

   Solution:

   (a) Since the hash index is on the candidate key, $\sigma_{A=const}(\mathbf{r})$ has at most one tuple. Therefore, the cost is 1.2 (searching the index) + 1(retrieving data). If the index is integrated then the cost is just 1.2.

   Note that the fact that even though the hash index is unclustered, we are not paying the price, because the index is on a candidate key.

   (b) Since B has 1,000 distinct values in **r**, there are about 5 tuples per value. Therefore $\sigma_{B=const}(\mathbf{r})$ is likely to retrieve 5 tuples. Because the index is clustered and because there are 5 tuples per page, the result fits in 1 page.

   Therefore, the cost is depth of the tree + 1.

   (c) Since C has 500 values, the selection is likely to produce 10 tuples (5000/50). Pointers to these tuples will be in the same or adjacent leaf pages of the B^+ tree. We conservatively estimate that these tuples will occupy 2 pages (index entries are typically much smaller than the data file records). Thus, the cost of retrieving all the pointers is 3 (to search the B^+ tree for the first page of pointers in the index) + 1 (to retrieve the second page of the index) = 4.

   Since the index is unclustered, each of the 10 tuples in the result can be in a separate page of the data file and its retrieval may require a separate I/O. Thus, the cost is 4+10 = 14.

   (d) Since we do not project out the candidate key, the projection will have the same number of tuples as the original. In particular, there will be no duplicates and no sorting will be required.

   The output will be about 2/3 of the original size assuming that all attributes contribute equally to the tuple size. Since the original file has 5,000/5=1000 blocks, the cost of the operation is 1,000(scan of the file) + 2/3*1,000 (cost of writing out the result).

3. Write down the sequence of steps needed to transform $\pi_A((\mathbf{R} \bowtie_{B=C} \mathbf{S}) \bowtie_{D=E} \mathbf{T})$ into $\pi_A((\pi_E(\mathbf{T}) \bowtie_{E=D} \pi_{ACD}(\mathbf{S})) \bowtie_{C=B} \mathbf{R})$. List the attributes that each of the schemas **R**, **S**, and **T** *must* have and the attributes that each (or some) of these schemas must *not* have in order for the above transformation to be correct.

   Solution:

   - **R** must have: B (because of the join)
   - **S** must have: ACD (because of π_{ACD})
   - **T** must have: E (because of π_E)
   - These schemas should not have identically named attributes, because otherwise it will not be clear which of the two identically named attributes will be renamed in the joins. In particular, **T** should not have A and C, because $\pi_{ACD}(\mathbf{S})$ clearly suggests that it is expected that **S** will have the attribute A that will survive for the outermost projection π_A to make sense, and the attribute C, which should survive in order for the join with **R** to make sense.

   (a) Associativity of the join: $\pi_A(\mathbf{R} \bowtie_{B=C} (\mathbf{S} \bowtie_{D=E} \mathbf{T}))$

   (b) Commutativity of the join: $\pi_A((\mathbf{S} \bowtie_{D=E} \mathbf{T}) \bowtie_{C=B} \mathbf{R})$

   (c) Commutativity of the join: $\pi_A((\mathbf{T} \bowtie_{E=D} \mathbf{S}) \bowtie_{C=B} \mathbf{R})$

   (d) Pushing projection π_A to the first operand of the join: $\pi_A(\pi_{AC}((\mathbf{T} \bowtie_{E=D} \mathbf{S})) \bowtie_{C=B} \mathbf{R})$

   (e) Pushing projection to the first operand in the innermost join: $\pi_A(\pi_{AC}((\pi_E(\mathbf{T}) \bowtie_{E=D} \mathbf{S})) \bowtie_{C=B} \mathbf{R})$. This is possible if AC are the attributes of **S** and not of **T**.

   (f) Pushing projection to the second operand in the innermost join: $\pi_A(\pi_{AC}((\pi_E(\mathbf{T}) \bowtie_{E=D} \pi_{ACD}(\mathbf{S}))) \bowtie_{C=B} \mathbf{R})$.

   (g) Reverse of pushing a projection: $\pi_A((\pi_E(\mathbf{T}) \bowtie_{E=D} \pi_{ACD}(\mathbf{S})) \bowtie_{C=B} \mathbf{R})$

4. Consider the following schema, where the keys are underlined:

   ITEM(Name,Category)
   STORE(Name, City, StreetAddr)
   TRANSACTION(ItemName, StoreName, Date)

   Here a tuple $(i, s, d) \in$ TRANSACTION denotes the fact that item i bought at store s on date d.)

   Consider the following query:

$$\pi_{\texttt{Category,City}}(\sigma_{\texttt{Date='2004-12-15' AND City='NYC'}}(\\ \text{ITEM} \bowtie_{\texttt{Name=ItemName}} \text{TRANSACTION} \bowtie_{\texttt{StoreName=Name}} \text{STORE} \\ ))$$

   Show the three "most promising" relational algebra expressions that the query optimizer is likely to consider; then find the most efficient query plan and estimate its cost.

   Assume 50 buffer pages and the following statistics and indices:

   - ITEM: 50,000 tuples, 10 tuples/page.
     Index: Unclustered hash on Name.
   - STORE: 1,000 tuples, 5 tuples/page; 100 cities.
     Index1: Unclustered hash index on Name.
     Index2: Clustered 2-level B^+ tree on City.
   - TRANSACTION: 500,000 tuples, 25 tuples/page; 10 items bought per store per day. The relation stores transactions committed over a 50 day period.
     Index: 2-level clustered B^+ tree on the pair of attributes StoreName, Date.

Solution: The optimizer will consider the fully pushed and the two partially pushed expressions:

$$\pi_{\texttt{Category,City}}(\text{ITEM} \bowtie_{\texttt{Name=ItemName}} \sigma_{\texttt{Date='2004-12-15'}}(\text{TRANSACTION}) \bowtie_{\texttt{StoreName=Name}} \sigma_{City='NYC'}(\text{STORE}))$$

$$\pi_{\texttt{Category,City}}(\text{ITEM} \bowtie_{\texttt{Name=ItemName}} \sigma_{\texttt{Date='2004-12-15'}}(\text{TRANSACTION} \bowtie_{\texttt{StoreName=Name}} \sigma_{City='NYC'}(\text{STORE})))$$

$$\pi_{\texttt{Category,City}}(\text{ITEM} \bowtie_{\texttt{Name=ItemName}} \sigma_{\texttt{City='NYC'}}(\sigma_{\texttt{Date='2004-12-15'}}(\text{TRANSACTION}) \bowtie_{\texttt{StoreName=Name}} \text{STORE}))$$

The most efficient query plan would be to use the last partially pushed expression with the join order reversed:

$$\pi_{\texttt{Category,City}}(\text{ITEM} \bowtie_{\texttt{Name=ItemName}} \sigma_{\texttt{Date='2004-12-15'}}(\sigma_{City='NYC'}(\text{STORE}) \bowtie_{\texttt{StoreName=Name}} \text{TRANSACTION}))$$

In addition, we can combine the selection of `Date` with the inner join, as explained below.

The costs are computed as follows:

(a) $\sigma_{City='NYC'}$(STORE) – selects 10 tuples (2 pages) using the 2-level B^+ tree index. Cost: 2+2=4.

(b) Join the result with TRANSACTION on `StoreName` using the 2-level B^+ tree index on TRANSACTION. Moreover, here we can combine the index nested loops join with selection on `Date` as follows. Note that the index is on the pair of attributes `StoreName Date`, while the join is only on the first attribute. However, we can combine each index search that is performed to compute the join with $\sigma_{\texttt{Date}}$. In this combination, every index search will be on the full index `StoreName Date` rather than just a prefix `StoreName`.

Since $\sigma_{City='NYC'}$(STORE) has 10 tuples, 10 full-index selections on TRANSACTION will be performed during the combined join/selection computation. According to the statistics, 10 items are sold per store per day. Therefore, each selection will bring 10 tuples, 1 page. It takes 3 I/Os to bring those tuples. Since we need to perform 10 selections, the cost is 3*10=30 and the result has 100 tuples.

(c) Join the result with ITEM using the unclustered hash index on `Name`. Since `Name` is a key, each hash brings 1 tuple and costs 1.2 (the cost of the hash) + 1 (the cost of retrieving the matching tuple) = 2.2 I/Os. If we do it 100 times for each tuple in the result of (2), we will spend 220 I/Os.

Therefore, the total is 4+30+220=254 I/Os.

5. Consider the following schema, where the keys are underlined:

   VIEWERS(Id,Age)
   MOVIE(Title, Director)
   HASSEEN(ViewerId, Title)

   This schema represents a survey of viewers of different movies.

   Consider the following query:

$$\pi_{\texttt{Title}}(\sigma_{\texttt{Director='JohnDoe' AND 25<Age<30}}\ (\text{VIEWERS} \bowtie_{\texttt{Id=ViewerId}} \text{HASSEEN} \bowtie_{\texttt{Title}} \text{MOVIE}))$$

   Show the three "most promising" relational algebra expressions that the query optimizer is likely to consider; then find the most efficient query plan and estimate its cost.

   Assume 62 buffer pages and the following statistics and indices:

   - VIEWERS: 30,000 tuples, 10 tuples/page.
     Index #1: Clustered 2-level B^+-tree index on `Age`.
     Index #2: Unclustered hash index on `Id`.
     The survey was conducted among viewers 21 to 60 years old.
   - MOVIE: 1,000 tuples, 5 tuples/page.
     Index: Unclustered hash index on `Director`.
     About 100 directors.
   - HASSEEN: 500,000 tuples, 50 tuples/page; an average viewer gave responses about 17 different films.
     Index #1: Unclustered hash index on `Title`
     Index #2: 2-level clustered B^+ tree on `ViewerId`.

Solution:

Plan 1. (second-best)

$$\pi_{\texttt{Title}}(\\ \quad (\sigma_{25<\texttt{Age}<30}(\textsc{Viewers}) \bowtie_{\texttt{Id=ViewerId}} \textsc{HasSeen}) \\ \quad \bowtie_{\texttt{Title}} \sigma_{\texttt{Director='JohnDoe'}}(\textsc{Movie}) \\ )$$

Select Viewers on Age: 300 pages (selection is on 4 ages out of 40 ages in the sample), 302 I/Os since the index is 2-level clustered B^+ tree.

Try to merge-join the result with HasSeen: Sort the result on `Id`: $2*300*\log_{59}300 = 4*300 = 1{,}200$. In the last stage of the sort merge, do merge with HasSeen, which is already sorted on `ViewerId`. This will require a scan of HasSeen, 10,000 I/Os. Then we need to join with selection on Movie — seems too expensive, so let's try another route.

Instead of the merge-join, let's try to use index-nested loops join using the clustered B^+ tree index on HasSeen.`ViewerId`. $\sigma_{25<\texttt{Age}<30}(\textsc{Viewers})$ has 3,000 tuples, so we need to use the index on `HasSeen` 3000 times. If we cache the root of that index, then the cost will be 1 (to cache the root) + 2*3000 = 6,001 I/Os.

Select Movie on `Director`: About $\lceil 10 + 1.2 \rceil = 12$ I/O.

To join with $\sigma_{\texttt{Director='JohnDoe'}}(\textsc{Movie})$, use block-nested loops. The strategy is the following. Compute $\sigma_{\texttt{Director='JohnDoe'}}(\textsc{Movie})$ *before* computing the join $\sigma_{25<\texttt{Age}<30}(\textsc{Viewers}) \bowtie_{\texttt{Id=ViewerId}} \textsc{HasSeen}$. Keep $\sigma_{\texttt{Director='JohnDoe'}}(\textsc{Movie})$ in the buffer while computing the above join. This will allow the result of the join to be pipelined into the final join with $\sigma_{\texttt{Director='JohnDoe'}}(\textsc{Movie})$ and compute the latter join at no additional cost.

The total cost is: 6001+302+12=6315.

Plan 2 (best).

$$\pi_{\texttt{Title}}(\\ \quad \sigma_{25<\texttt{Age}<30} (\\ \quad\quad (\sigma_{\texttt{Director='JohnDoe'}}(\textsc{Movie}) \bowtie_{\texttt{Title}} \textsc{HasSeen}) \\ \quad\quad \bowtie_{\texttt{Id=ViewerId}} \textsc{Viewers} \\ \quad\quad) \\ )$$

Select Movie on `Director`: About $\lceil 10 + 1.2 \rceil = 12$ I/O.
Join with HasSeen on `Title` using unclustered hash index on HasSeen.`Title`. Since there are 1000 films and 500,000 tuples in HasSeen, there are 500 tuples per title, so this will require 10*(1.2+500)=12+5000=5012 I/Os. The result has 5,000 tuples and the `ViewerId` attribute mentions about $5000/17 \approx 300$ different viewers. Since we have 60 buffers, we can cache all these viewers in main memory, so pipelining is possible and we do not need to hash twice (into ViewerId) on the same viewer Id. While pipelining, use the hash index on Viewer.`Id` to

join with VIEWERS: another (1.2+1)*300 = 660 I/Os.
Total: 12+5012+660=5,684 I/Os.

Plan 3 (distant third)

$$\pi_{\texttt{Title}}(\\ (\sigma_{\texttt{Director='JohnDoe'}}(\text{MOVIE}) \bowtie_{\texttt{Title}} \text{HASSEEN}) \\ \bowtie_{\texttt{Id=ViewerId}} \sigma_{\texttt{25<Age<30}}(\text{VIEWERS}) \\ )$$

The evaluation is like in Plan 2, but we apply selection to VIEWERS. This eliminates the indices on VIEWERS and so we must use an expensive blocks nested loops join at the last stage. It is expensive because the join MOVIE) $\bowtie_{\texttt{Title}}$ HASSEEN) is large (5000 tuples and at least 1000 pages) and is pipelined. To avoid its materialization on disk, this relation should be in the outer loop of the block nested loops join. The relation $\sigma_{\texttt{25<Age<30}}$(VIEWERS) is 300 pages long and is used in the inner loop. Since the buffer has 60 pages, it would require reading that latter 300-page relation at least 1000/60 times, i.e., more than 300*16=4,800 I/Os. This is more expensive than the 600 I/Os that we needed to join with VIEWERS in Plan 2. (Using sort-merge to do that last join is even more expensive: $2 * log_{59}1000 \ + \ 2 * 300 * log_{59}300$, which is more than 6,000+1,200= 7,200.)

6. Consider the following relations that represent part of a real estate database:

   AGENT(Id, AgentName)
   HOUSE(Address, OwnerId, AgentId)
   AMENITY(Address, Feature)

   The AGENT relation keeps information on real estate agents, the HOUSE relation has information on who is selling the house and the agent involved, and the AMENITY relation provides information on the features of each house. Each relation has its keys underlined.

   Consider the following query:

```
SELECT  H.OwnerId, A.AgentName
FROM    HOUSE H, AGENT A, AMENITY Y
WHERE   H.Address=Y.Address AND A.Id = H.AgentId
                AND Y.Feature = '5BR' AND H.AgentId = '007'
```

   Assume that the buffer space available for this query has 5 pages and that the following statistics and indices are available:

   - AMENITY:

     10,000 records on 1,000 houses, 5 records/page
     Clustered 2-level B^+ tree index on `Address`
     Unclustered hash index on `Feature`, 50 features

   - AGENT:

     200 agents with 10 tuples/page
     Unclustered hash index on `Id`

   - HOUSE:

     1,000 houses with 4 records/page
     Unclustered hash index on `AgentId`
     Clustered 2-level B^+ tree index on `Address`

   Answer the following questions (and explain how you arrived at your solutions).

   (a) Draw a fully pushed query tree corresponding to the above query.

   Solution:

   See Figure 1.

   (b) Find the best query plan to evaluate the above query and estimate its cost.

   Solution:

   We could join HOUSE with AGENT or AMENITY, but in any case it is clear that we should first select HOUSE on `AgentId`, because of the large reduction in size: There are 200 agents, 1000 houses, so agent 007 must be handling about 5 houses. At 4 records per page, this would occupy 2 pages.

   Because the index on `AgentId` is unclustered, it would take 1.2 I/Os to find the bucket and 5 I/Os to fetch the relevant pages: 6.2 I/Os in total.

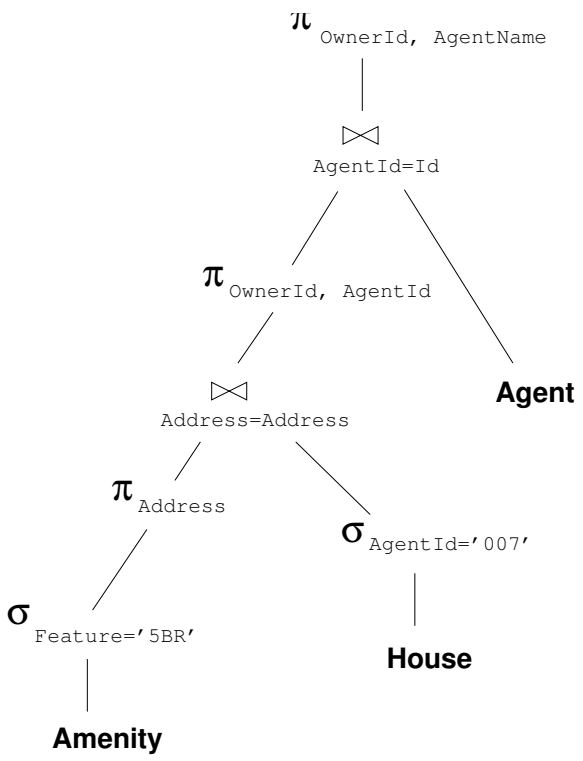

Figure 1:

Next we can join with AGENT, which would take 1.2 page I/Os to search the index, since AGENT has an unclustered index on `Id`, plus 1 I/O to fetch the page — 2.2 I/Os in total. This will still result in 5 tuples, but the tuples will be about 50% larger (AGENT has 10 tuples/page, while HOUSE only 4). However, we can project out `Id` and `AgentId`, which will bring the tuple size to about the size of the tuples in HOUSE. So, the join will still occupy a little over a page. We keep the result of the join in the main memory buffer.

Finally, we join the result with AMENITY. Since the statistics indicate that there are about 10 amenities per house, it doesn't make much sense to select AMENITY on `Feature`: the size will go down by the factor of 10 at a very high price (unclustered hash or scan of 2,000 blocks of the AMENITY relation), and we will loose the index on `Address` — the attribute used in the join.

So, the best way to do the join is to use index-nested loops join using the clustered index on the attribute `Address` of AMENITY. It would take 2 I/Os to search the B^+ tree index for each of the 5 tuples in the result of the previous join (i.e., 2*5; *if we cache the top level* of the B^+ tree then this search would take 2+4=6 I/Os). The number of tuples retrieved would be 50 (10 features per house * 5 houses), which occupies 10 pages. Therefore, the total needed to retrieve the matching tuples in AMENITY is 16 I/Os.

Note that we still have enough room in the buffer. The expected size of the join is 50 tuples (5*10), which is too large for our 5 page buffer. However, we can also select on `Feature='5BR'` on the fly, reducing the size to about 5 tuples, each about twice the size of the tuples in HOUSE. We can also project (on the fly) on `OwnerId` and `AgentName` further reducing the size. Thus, we will need 2 pages in the buffer for the result of the

join of HOUSE and AGENT, one page for the input buffer that is needed for reading the matching tuples of AMENITY, and two to store the final result. This fits in the available 5 buffer pages.

In total, thus, the query will take 6.2 + 2.2 + 16 = 24.4 I/Os.

(c) Find the next-best plan and estimate its cost.

Solution:

Similar, except that what is left of HOUSE is first joined with AMENITY. The number of I/Os is going to be the same, so this is also a best plan.

The next plan after that would be to do something silly, like joining HOUSE and AGENT using nested loops. Since AGENT occupies 20 blocks, this can be done in 20 I/Os. Then we could proceed to join with AMENITY.

7. The following is partial schema for a public library database, where the keys are underlined:

   CUSTOMER(CustId,Name)
   BORROWINGS(CustId, ItemId, BorrowDate, DueDate)
   ITEM(ItemId, Author, Title)

   Consider the following query:

   $$\pi_{\texttt{CustId}}(\sigma_{\texttt{Name='JohnDoe' AND 2006/05/19<BorrowDate<2006/05/25 AND Author='JoePblic'}}(\text{CUSTOMER} \bowtie \text{BORROWINGS} \bowtie \text{ITEM}))$$

   Show three most promising relational algebra expressions that the query optimizer is likely to consider; then find the most efficient query plan and estimate its cost.

   Assume 10 buffer pages and the following statistics and indices:

   - BORROWINGS: 10 tuples/page.
     Index #1: Unclustered 2-level B^+-tree index on the attribute sequence CustId,BorrowDate, ItemId.
     Index #2: Clustered hash index on ItemId.
     A customer can borrow up to 2 items on any given day.
     On average, an item has been borrowed 50 times.
   - CUSTOMER: 5 tuples/page.
     Index #1: Unclustered hash index on Name.
     Index#2: Clustered 2-level B^+-tree index on CustId.
     On average, two different people have the same name.
   - ITEM: 10 tuples/page.
     Index #1: Clustered hash index on Author.
     Index #2: Unclustered hash index on ItemId.
     On average, a person can be an author for 10 items.

   Solution: The following is the most efficient query plan. Other promising plans that the optimizer would consider are obtained from this by pushing $\sigma_{\texttt{2006/05/19<BorrowDate<2006/05/25}}$ down to BORROWINGS and by pulling out $\sigma_{\texttt{Name='JohnDoe'}}$ or $\sigma_{\texttt{Author='JoePublic'}}$.

   $$\pi_{\texttt{CustId}}(\sigma_{\texttt{2006/05/19<BorrowDate<2006/05/25}}(\sigma_{\texttt{Name='JohnDoe'}}(\text{CUSTOMER}) \bowtie \text{BORROWINGS}) \bowtie \sigma_{\texttt{Author='JoePublic'}}(\text{ITEM}))$$

   The first join is index-only join, so it does not matter that the index on CustId,BorrowDate, ItemId is unclustered.

   Cost of $\sigma_{Name='JoePublic'}(Customer)$: 1.2 (hash index on Name) + 2 (to retrieve the two people with that name) = 3.2 (actually 4, since we need to round up).

   Compute $\sigma_{\texttt{2006/05/19<BorrowDate<2006/05/25}}(\sigma_{\texttt{Name='JohnDoe'}}(\text{CUSTOMER}) \bowtie \text{BORROWINGS})$ After computing the inner selection, we can compute the join using index-only join, i.e., $\sigma_{Name='JoePublic'}(Customer)$

can be joined with just the index of BORROWINGS on the attributes `CustId`, `BorrowDate`, `ItemId`. While computing this join, we can also apply the selection $\sigma_{\texttt{2006/05/19<BorrowDate<2006/05/25}}$ using the same index.

There are at most $2(customers) * 2(items) * 5(days) = 20$ tuples in the result, which means at most 3 pages to access (as before, they may cross a page boundary. Since there are two levels in the B^+ tree that implements the index on BORROWDATE, the cost is 2 (levels) + 3 pages = 5I/Os.

Cost of $\sigma_{\texttt{Author='JoePublic'}}$(ITEM): 1.2 (hash on Author) + 2 (the index is clustered, so all selected tuples are close together; but they may span page boundary and reside in 2 different pages) = 3.2 = 4 (rounding up to a whole number of disk accesses).

The final join is in memory, so the total is 4+4+5-13 I/O.

Problems for Chapter 12: Database Tuning

1. Consider the following relational schema:

   STUDENT(Id,Name,Major)
   TOOK(StudId,Course)

   where Id is the primary key in STUDENT. Consider the query

   ```
   SELECT *
   FROM STUDENT S, TOOK T
   WHERE S.Id = T.StudId AND T.Course = 'CS305' AND S.Major = 'EE'
   ```

   (a) Write a relational algebra expression that is equivalent to this query.

   (b) Suggest the indexes that can be added to this database in order to improve the performance of this query. Indicate whether these indices should be clustered or not. Explain your reasoning briefly.

   Solution:

   (a) $\sigma_{\texttt{Course='CS305'}}(\texttt{Took}) \bowtie_{\texttt{StudId=Id}} \sigma_{\texttt{Major='EE'}}(\texttt{Student})$

   (b) Clustered index on Course in TOOK, since it will facilitate the first selection. A clustered index on Major in STUDENT might also help, since it can facilitate the second selection. Note that although Id is a primary key in STUDENT, it might have an unclustered (rather than clustered) index, because numeric single-attribute keys often have unclustered hash indices.

   We could do away with the clustered index on Major, since we could also do the join using the index on Id, which exists because Id is the primary key.

2. The table Employee(Id, Name, DeptId, Salary) has `Id` as its primary key. What index would you choose to enhance the performance of the following statement?

```
SELECT E.Name
FROM Employee E
WHERE E.Salary > 100000 AND E.DeptName = 'accounting'
```

Solution:

The index on the primary key is of no help. An additional index on either `Salary` or `DeptName` is needed depending on which is more selective. If the number of employees in accounting is smaller than the number earning over $100000 then a hash or B^+ tree index on `DeptName` would be appropriate. Performance can be further improved by making the index clustered, so that all employees in the same department are in the same hash bucket or are consecutive if a B^+ structure is used. In that case the index on Id must by unclustered, but that should not imply a performance penalty since queries involving `Id` will generally not involve a range search and hence will return a single row. If accounting is a large department then a B^+ tree index on `Salary` would be appropriate since a range search is needed. The same considerations with respect to clustering apply.

3. The table Employee(Id, Name, DeptId, Salary, $\cdots$) has `Id` as its primary key. Describe a situation that might justify the use of an index covering strategy.

   Solution:

   Assume that the table has a large number of attributes, so that each row occupies a significant amount of space. Assume also that a clustered index exists on certain attributes to enhance the performance of a particular query. For example, a clustered index on `Salary` might be required if the query

   ```
   SELECT E.Name
   FROM Employee E
   WHERE E.Salary > :sal
   ```

   were frequently executed. Suppose another frequently asked query is

   ```
   SELECT E.Name
   FROM Employee E
   WHERE E.DeptId = :dept
   ```

   Since the number of employees in a department might be large it is desirable to support the query with a clustered index on `DeptId`, but this is not possible since one clustered index already exists. An unclustered B^+ tree index on (`DeptId, Name`) contains all the information needed to respond to the query without consulting the table itself. Leaf level entries are sorted on `DeptId` and hence can be efficiently retrieved. Note that since the number of characters needed to store the values of these attributes is small, the index itself is small and only a few pages will need to be retrieved to satisfy the query.

4. The SELECT statement

```
SELECT A
FROM T
WHERE P
```

accesses a table, T, with attributes A, B, C and K, where K is the primary key, and P is a predicate. The values of all attributes are randomly chosen over the integer domain. Since the table is updated frequently, there is a limitation of at most two indexes. In each of the following parts choose the indexes (type, search key, clustered or unclustered) to optimize the SELECT statement and maintain the primary key constraint when the value of P is as specified. In each case give the query plan you expect the DBMS to choose for the SELECT and the reason you think that plan is best. Do not use index covering to reduce querying to index-only queries.

Note: since K is the primary key, it has an index. So, we only have to decide what kind of index (clustered/unclustered, etc.) and what should be the other indices.

(a) P is (B = 10)

Solution:

clustered index on B; unclustered on K
search on B=10, scan (B^+ tree) or search bucket (hash) for all satisfying rows

(b) P is (B > 10)

Solution:

clustered B^+ tree on B, unclustered on K
rows are ordered on B, search for B=10 and scan from that point

(c) P is (B > 10 AND C = 5)

Solution:

Fewer rows satisfy C=5 than B>10 (values are randomly chosen). Hence use clustered index on C, unclustered on K; hash or B^+ tree OK for both.
Fetch rows satisfying C=5 and return those having B>10

(d) P is (B > 10) and the WHERE clause is followed by the clause ORDER BY A

Solution:

Clustered B^+ tree on A, unclustered hash or B^+ on K. Scan entire file, discard rows return rows with B>10. This is better than clustering on B since roughly half of the rows satisfy B>10 and they would require sorting.

(e) P is (B > 10 AND C = 5)

Solution:

Fewer rows satisfy C=5 than B>10 (values are randomly chosen). Hence use clustered index on C, unclustered on K; hash or B^+ tree OK for both.
Fetch rows satisfying C=5 and return those having B>10

(f) P is (B > 10) and the WHERE clause is followed by the clause ORDER BY A

Solution:

Clustered B^+ tree on A, unclustered hash or B^+ on K. Scan entire file, discard rows return rows with B>10. This is better than clustering on B since roughly half of the rows satisfy B>10 and they would require sorting.

5. Consider a database with two tables: EMP, with attributes `empName` and `divName` and DIV with attributes `divName` and `building`. You are required to write a select statement that lists the names of all employees in divisions that have space in building A. (A division might have space in several buildings.)

   (a) Write the statement using a join on the two relations.

   Solution:

```
SELECT E.empName
FROM  EMP E,  DIV D
WHERE E.divName = D.divName  AND D.building = 'A'
```

   (b) Write the statement in a form that uses a subquery, but no join.

   Solution:

```
SELECT E.empName
FROM  EMP E
WHERE E.divName  IN (
        SELECT  D.divName
        FROM  DIV D
        WHERE D.building = 'A')
```

   (c) Assuming no useful indexes exist, comment on the efficiency of executing the two forms of the query. For example, under what conditions (e.g., on the numnber of rows in each relation, the number of divisions with offices in building A, the number of employees in a division) would you expect one form of the query to outperform the other.

   Solution:

   Assume n pages in EMP and m pages in DIV. A query plan for the first statement might do a scan of DIV for each page of EMP, requiring $n \times m$ page transfers.

   Since the subquery is not correlated, a query plan for the second might first evaluate the subquery to identify divisions having space in building A. The scan requres m page transfers and results in an intermediate relation of r pages where $max(r) = m$. It then does $n \times r$ page transfers to combine rows in EMP with the intermediate relation. Hence the total number of operations is $m + n \times r$ and if r is much less than m - which is likely - the second statement can be evaluated more efficiently.

   (d) In a more general situation, the identity of the building will be input as a parameter. What measure might you take (denormalization, use of an index) to speed the execution of first statement?

   Solution:

   Denormalizing by adding a building attribute to EMP doesn't work, since an employee in building B might be working for a division that also has space in building A. Indexing DIV on `divName` allows the use of an index-nested loop join. The scan of DIV can be avoided to form the intermediate relation.

Problems for Chapter 13: Relational Calculus, Deductive Databases, and Visual Query Languages

1. Consider the following relational schema about real estate brokers. Keys are underlined.

   BROKER(Id, Name)
   SOLD(BrokerId, HouseId, Price)
   LOCATION(HouseId, Town)

   Write the following query in **domain** relational calculus: "*Find all brokers (Id, Name) who did not sell even a single house in Stony Brook.*"

   Solution:

   ```
   {Id, Name | BROKER(Id,Name) AND
            NOT(∃HouseId ∃Price (
                    SOLD(Id,HouseId,Price) AND
                    LOCATION(HouseId, "Stony Brook") )) }
   ```

2. Translate the following relational algebraic expression into **tuple** relational calculus.

$$\pi_{\texttt{Id}}(\textsc{Broker} \bowtie_{\texttt{Id = BrokerId}} \textsc{Sold} \bowtie \sigma_{\texttt{Town} \neq \texttt{'StonyBrook'}}(\textsc{Location}))$$

The schema is as follows:

```
Broker(Id, Name)
Sold(BrokerId, HouseId, Price)
Location(HouseId, Town)
```

Solution:

```
{B.Id | Broker(B) AND ∃S∈Sold ∃L∈Location(
          S.BrokerId = B.Id AND S.HouseId = L.HouseId
               AND NOT (L.Town = "Stony Brook")) }
```

3. Write the following query using TRC and DRC: Find the names of all brokers who have made money in all accounts assigned to them. Assume the following schema:

   BROKER(Id,Name)
   ACCOUNT(Acct#, BrokerId, Gain)

   Solution:

   TRC: $\{B.\texttt{Name} \mid \text{BROKER}(B) \text{ AND } \forall A \in \text{ACCOUNT}\,(A.\texttt{BrokerId} = B.\texttt{Id} \rightarrow A.\texttt{Gain} > 0)$

   DRC: $\{\texttt{Name} \mid \exists \texttt{BrokerId}\,(\text{BROKER}(\texttt{BrokerId}, \texttt{Name}) \text{ AND } \forall \texttt{Acc\#}\,\forall \texttt{Gain}\,(\text{ACCOUNT}(\texttt{Acc\#}, \texttt{BrokerId}, \texttt{Gain}) \rightarrow \texttt{Gain} > 0))\}$

4. Express the following query in TRC and DRC: Find all courses in department MGT that were taken by all students. Assume the following schema:

   COURSE(CrsCode,DeptId,CrsName,Descr)
   TRANSCRIPT(StudId,CrsCode,Semester,Grade)
   STUDENT(Id,Name,Status,Address)

   Solution:

   TRC: {C.CrsCode, C.CrsName | COURSE(C) AND C.DeptId = 'MGT' AND
   $\forall S \in$ STUDENT $\exists R \in$ TRANSCRIPT
   (R.StudId = S.Id AND R.CrsCode = C.CrsCode) }

   DRC: {CrsCode, CrsName | $\exists$Descr (COURSE(CrsCode,'MGT',CrsName,Descr))
   AND $\forall$Id $\in$ STUDENT.Id $\exists$Semester$\exists$Grade
   (TRANSCRIPT(Id,CrsCode,Semester,Grade)) }

5. Consider the following relational schema where the keys are underlined:

   ACTOR(Id, Name)
   MOVIE(Id, Title, DirectorName)
   PLAYED(ActorId, MovieId)

   Write the following query in **tuple** relational calculus: "*Find all actors (Id, Name) who played in every movie produced by the director Joe Public*."

   Solution:

$$\{A.\texttt{Id}, A.\texttt{Name} \mid \textsc{Actor}(A) \text{ AND } \forall M \in \textsc{Movie}(M.\texttt{DirectorName} = \text{'Joe Public'} \rightarrow \exists P \in \textsc{Played}(M.\texttt{Id} = P.\texttt{MovieId} \text{ AND } A.\texttt{Id} = P.\texttt{ActorId})) \}$$

6. Consider a relation DIRECTFLIGHT(StartCity, DestinationCity) that lists all direct flights among cities. Use the recursion facility of SQL:1999 to write a query that finds all pairs $\langle city_1, city_2 \rangle$ such that there is an *indirect* flight from $city_1$ to $city_2$ with at least two stops in-between.

   Solution:

   The simplest way to do this is to compute INDIRECTFLIGHT similarly to INDIRECTPREREQVIEW:

```
CREATE RECURSIVE VIEW INDIRECTFLIGHT(From, To) AS
        SELECT * FROM DIRECTFLIGHT
        UNION
        SELECT D.StartCity, I.To
        FROM DIRECTFLIGHT D, INDIRECTFLIGHT I
        WHERE D.DestinationCity = I.From
```

   Then we can compute all flights with just one stop — FLIGHT1 — and then subtract DIRECTFLIGHT and FLIGHT1 from INDIRECTFLIGHT.

   One might be tempted to first create a recursive view INDIRECTFLIGHT2(From, TO, NumberOfStops) and then select the flights with NumberOfStops > 1.

```
CREATE RECURSIVE VIEW INDIRECTFLIGHT2(From, To, Stops) AS
        SELECT D.StartCity, D.DestinationCity, 0
        FROM DIRECTFLIGHT D
        UNION
        SELECT D.StartCity, I.To, I.Stops+1
        FROM DIRECTFLIGHT D, INDIRECTFLIGHT2 I
        WHERE D.DestinationCity = I.From
```

   However, this recursive definition has a problem: because we keep incrementing the number of stops with each iteration, the evaluation process will not terminate. (Check that the termination condition will never be true!)

7. Consider a relation schema of the form

   PERSON(Name, Hobby)

   which, for each person, specifies the hobbies that the person has. For instance, such a relation might have the following content:

   ```
   JohnDoe   chess
   JohnDoe   jogging
   MaryDoe   jogging
   MaryDoe   hiking
   JoePublic reading
   ```

   Write the following queries in Datalog:

   (a) Find all people whose sets of hobbies are contained within the set of hobbies of MaryDoe. This query involves a nontrivial use of double negation.

   (b) Find all people who can be linked to JohnDoe via a chain of hobbies. This includes all those who have at least one common hobby with JohnDoe; all those who have a common hobby with the people who have a common hobby with JohnDoe; etc. The chain of hobbies can be arbitrarily long. This query involves recursion.

   Solution for (a):

   ```
   Answer(?P) :- Person(?P,?SomeHobby),
                 not PersonWithHobbyThatMaryDoesNotHave(?P).
   PersonWithHobbyThatMaryDoesNotHave(?P) :-
                 Person(?P,?H), not Person(MaryDoe,?H).
   ```

   Solution for (b):

   ```
   Answer(?P) :- LinkedByHobbyChain(JohnDoe,?P).
   LinkedByHobbyChain(?P1,?P2) :- MutualHobby(?P1,?P2).
   LinkedByHobbyChain(?P1,?P2) :- MutualHobby(?P1,?P3),
                                  LinkedByHobbyChain(?P3,?P2).
   MutualHobby(?P1,?P2) :- Person(?P1,?H), Person(?P2,?H).
   ```

Problems for Chapter 14: Object Databases

1. Use the following partially defined schema to answer the questions below.

```
CREATE TYPE StudentType AS (
  Id INTEGER,
  Name CHAR(20),
  ...
  Transcript TranscriptType MULTISET
)
CREATE TYPE TranscriptType AS (
  Course REF(CourseType) SCOPE Course,
  ...
  Grade GradeType
)
CREATE TABLE Student OF StudentType;
CREATE TABLE Transcript OF TranscriptType;
```

The type CourseType is defined as usual, with character string attributes such as `CrsCode`, `DeptId`, etc.

(a) Find all students who have taken more than five classes in the mathematics department.

Solution:

```
SELECT S.Name
FROM Student S
WHERE 5 < ( SELECT count(T.Course)
            FROM   Student S1, UNNEST(S1.Transcript) T
            WHERE  T.Course->DeptId = 'MAT'
                   AND  S1.Id = S.Id)
```

(b) Define the UDT GradeType used in the definition of TranscriptType above. Define the method `value()` for GradeType, which returns the grade's numeric value.

Solution:

```
CREATE TYPE GradeType AS (
   LetterValue CHAR(2) )
METHOD value() RETURNS DECIMAL(3);

CREATE METHOD value() FOR GradeType
RETURNS DECIMAL(3)
LANGUAGE SQL
   BEGIN
      IF LetterValue = 'A' THEN RETURN 4;
      IF LetterValue = 'A-' THEN RETURN 3.66;
      ... ... ...
   END
```

(c) Write a method for StudentType that, for each student, computes the average grade. This method can use the `value()` method that you constructed for the previous problem.

Solution:

```
CREATE METHOD avgGrade() FOR StudentType
RETURNS DECIMAL(3)
LANGUAGE SQL
BEGIN
    RETURN (
        SELECT avg(T.Grade.value())
        FROM   Student S, UNNEST(S.Transcript) T
        WHERE  S.Id = Id )
END
```

Note that `Id` in the above method definition is the `Id` attribute of the Student-object on which the method operates.

2. Consider the following schema:

   ACTOR(Id, Name)
   MOVIE(Id, Title, DirectorName)
   PLAYED(ActorId, MovieId)

   (a) Represent this schema using the object-relational extensions of SQL:1999/2003. Use set-valued attributes and object-oriented design.

   Solution:

```
CREATE TYPE PERSONTYPE AS (
   Id INTEGER,
   Name CHAR(30)
)
CREATE TYPE ACTORTYPE UNDER PERSONTYPE AS (
   PlaysIn REF( MOVIETYPE) MULTISET SCOPE MOVIES
)
CREATE TYPE MOVIETYPE AS (
   Id INTEGER,
   Title CHAR(100),
   Director REF( PERSON),
   Stars REF( ACTORTYPE) MULTISET SCOPE ACTORS
)
CREATE TABLE Actors OF ACTORTYPE
CREATE TABLE Movies OF MOVIETYPE
```

   (b) Use the schema constructed in subproblem (a) to formulate the following query using SQL:2003: *For every actor,* ac*, count the number of movies directed by Joe Schmo where the actor* ac *played a role.*

Solution:

```
SELECT A.Id, count(M.Title)
FROM ACTORS A, Movies M
WHERE A IN M.Stars AND M.Director->Name = 'Joe Schmo'
GROUP BY A.Id
```

3. (a) Consider the following enterprise. There are three basic types of entities: Sellers, Buyers, and Transactions. A seller object has an Id, Name, and a set of transactions he is involved in. A buyer object contains similar information. A transaction represents a deal between a single buyer and a single seller regarding transfer of ownership of one or more products. Use the object-relational extensions of SQL:1999/2003 to specify the appropriate schema. Use set-valued attributes and follow the principles of the object-oriented design.

   (b) Use the above schema to formulate the following query using SQL:1999/2003: *For every seller, show his Id, Name, and the number of buyers with whom the seller had a transaction.*

   Solution for (a):

```
CREATE TYPE PersonType AS (
  Name CHAR(50),
  Id   CHAR(9),
  PRIMARY KEY (Id)
)
CREATE TYPE BuyerType UNDER PersonType AS (
  Transactions REF( TransactionType)
               MULTISET SCOPE Transaction
)
CREATE TYPE SellerType UNDER PersonType AS (
  Transactions REF( TransactionType)
               MULTISET SCOPE Transaction
)
CREATE TYPE TransactionType AS (
  Id   CHAR(10),
  Date date,
  Buyer REF( BuyerType) SCOPE Buyer,
  Seller REF( SellerType) SCOPE Seller
)
CREATE TABLE Person of PersonType;
CREATE TABLE Buyer of BuyerType;
CREATE TABLE Seller of SellerType;
CREATE TABLE Transaction of TransactionType;
```

   Note: Putting `Transactions REF(TransactionType)` in PersonType is tempting, but is not right. This attribute has a different meaning in SellerType and BuyerType so, in effect, we need two attributes in case the same person is both a buyer and a seller. Putting this attribute in PersonType would give us only one attribute!

   Solution for (b):

```
SELECT S.Id, S.Name, count(DISTINCT T->Buyer->Id)
FROM Seller S, UNNEST(S.Transactions) T
GROUP BY S.Id, S.Name
```

Note that the variable `T` in the above is of type REF(TransactionType), so we have to use an arrow to get to the `Buyer` attribute. Likewise, `T->Buyer` is of type REF(BuyerType), so we need to use an arrow again in order to get to the Id attribute of BuyerType.

4. Consider an ACCOUNT class and a TRANSACTIONACTIVITY class in a banking system.

   (a) Posit ODMG ODL class definitions for them. The ACCOUNT class must include a relationship to the set of objects in the TRANSACTIONACTIVITY class corresponding to the deposit and withdraw transactions executed against that account.

   Solution:

```
class ACCOUNT {
    attribute Integer AcctId;
    relationship Set< PERSON> Owner;
    relationship Set< TRANSACTIONACTIVITY> Transactions
                 inverse TRANSACTIONACTIVITY::ActivityAccount;
}

class TRANSACTIONACTIVITY {
   // assume type is some integer code
    attribute enum TransType {deposit,withdraw} Type;
    attribute Float  Amount;
    attribute Date   ActivityDate;
    relationship ACCOUNT ActivityAccount
                   inverse ACCOUNT::Transactions;
}
```

   (b) Give an example of an object instance satisfying that description.

   Solution:

   Extent of class ACCOUNT:
   (#123, [12345, {#p4345, #p0987}, {#t435, #t8132}])

   Extent of class TRANSACTIONACTIVITY:
   (#t435, [withdraw, 58.34, 2001-3-4, #123])
   (#t8132, [deposit, 231.99, 2001-4-5, #123])

   Here #p4345, #p0987 denote object Ids of some unspecified PERSON-objects.

   (c) Give an example of an OQL query against that database, which will return the account numbers of all accounts for which there was at least one withdrawal of more than $10,000.

   Solution:

```
SELECT A.AcctId
FROM   ACCOUNTEXT A, A.Transactions T
WHERE T.Type = withdraw  AND T.Amount > 10000
```

5. (a) Use ODL of the ODMG standard to represent a schema for the following enterprise. There are three basic types of entities: Sellers, Buyers, and Transactions. A seller object has an Id, Name, and a set of transactions he is involved in. A buyer object contains similar information. A transaction represents a deal between a single buyer and a single seller regarding transfer of ownership of one or more products. The design must be object-oriented.

   (b) For the schema created in (a), formulate the following query using OQL: *For every seller, show the Id, Name, and the number of buyers with whom the seller had a transaction.*

Solution for (a):

```
Interface PersonInterface : Object {
   attribute String Id;
   attribute String Name;
}
Class Buyer : PersonInterface
   (extent BuyerExt keys Id): persistent
{
   relationship SET< Transaction> Transactions
                inverse Transaction::Buyer;
}
Class Seller : PersonInterface
   (extent SellerExt keys Id): persistent
{
   relationship SET< Transaction> Transactions
                inverse Transaction::Seller;
}
Class Transaction : Object
   (extent TransExt keys Id): persistent
{
   attribute String Id;
   attribute Date Date;
   relationship Buyer Buyer inverse Buyer::Transactions;
   relationship Seller Seller inverse Seller::Transactions;
}
```

Solution for (b):

```
SELECT S.Id, S.Name,
       count (SELECT DISTINCT T.Buyer.Id
              FROM S.Transactions T)
FROM SellerExt S
```

6. Write an OQL query that, for each major, computes the number of students who have that major. Use the following partial definition of STUDENT:

```
class STUDENT extends PERSON
  (extent STUDENTEXT)
  ...
  attribute Set<String> Major;
  relationship Set<Course> Enrolled;
               inverse COURSE::Enrollment;
  ...


class COURSE : Object
  ...
  relationship Set<Student> Enrollment;
               inverse STUDENT::Enrolled;
  ...
```

Solution:

```
SELECT Major: M,  count: count(S.Id)
FROM STUDENTEXT S, flatten(S.Major) M
GROUP BY M
```

We could also write this query using nested queries as follows:

```
SELECT DISTINCT Major: M,
              count: count( SELECT S2.Id
                            FROM STUDENTEXT S2
                            WHERE M IN S2.Major  )

FROM STUDENTEXT S, flatten(S.Major) M
```

Problems for Chapter 15: XML and Web Data

1. Specify a DTD appropriate for a document that contains data from both the COURSE table in Figure 4.34 and the REQUIRES table in Figure 4.35. Try to reflect as many constraints as the DTDs allow. Give an example of a document that conforms to your DTD.

   Solution:

   One good example of such a document is shown below. Note that it does not try to mimic the relational representation, but instead courses are modeled using the object-oriented approach.

```
<Courses>
   <Course CsrCode="CS315" DeptId="CS"
           CrsName="Transaction Processing" CreditHours="3">
      <Prerequisite CrsCode="CS305" EnforcedSince="2000/08/01"/>
      <Prerequisite CrsCode="CS219" EnforcedSince="2001/01/01"/>
   </Course>
   <Course>
      .....
   </Course>
   .....
</Courses>
```

   An appropriate DTD would be:

```
<!DOCTYPE Courses [
   <!ELEMENT Courses (Course*)>
   <!ELEMENT Course (Prerequisite*)>
   <!ATTLIST Course
               CrsCode ID #REQUIRED
               DeptId  CDATA #IMPLIED
               CrsName CDATA #REQUIRED
               CreditHours CDATA #REQUIRED >
   <!ATTLIST Prerequisite
               CrsCode IDREF #REQUIRED
               EnforcedSince CDATA #REQUIRED>
]>
```

2. Define the following simple types:

   (a) A type whose domain consists of lists of strings, where each list consists of 7 elements.
   Solution:

```
<simpleType name="ListsOfStrings">
   <list itemType="string" />
</simpleType>
<simpleType name="ListsOfLength7">
   <restriction base="ListsOfStrings">
        <length value="7"/>
   </restriction>
</simpleType>
```

   (b) A type whose domain consists of lists of strings, where each string is of length 7.
   Solution:

```
<simpleType name="StringsOfLength7">
   <restriction base="string">
      <length value="7"/>
   </restriction>
</simpleType>
<simpleType name="ListsOfStringsOfLength7">
   <list itemType="StringsOfLength7" />
</simpleType>
```

   (c) A type appropriate for the letter grades that students receive on completion of a course—A, A−, B+, B, B−, C+, C, C−, D, and F. Express this type in two different ways: as an enumeration and using the `pattern` tag of XML Schema.
   Solution:

```
<simpleType name="gradesAsEnum">
   <restriction base="string">
      <enumeration value="A"/>
      <enumeration value="A-"/>
      <enumeration value="B+"/>
      ..........
   </restriction>
</simpleType>
<simpleType name="gradesAsPattern">
   <restriction base="string">
      <pattern value="(A-?|[BC][+-]?|[DF])"/>
   </restriction>
</simpleType>
```

   In the `gradesAsEnum` representation, (...) represent complex alternatives of patterns separated by | (W3C has adopted the syntax of regular expressions used in Perl), [...] represent simple alternatives (of characters), and ? represents zero or one occurrence, as usual in regular expressions.

3. Write an XML schema specification for a simple document that lists stock brokers with the accounts that they handle and a separate list of the client accounts. The Information about the accounts includes the account `Id`, ownership information, and the account positions (*i.e.*, stocks held in that account). To simplify the matters, it suffices, for each account position, to list the stock symbol and quantity. Use `ID`, `IDREF`, and `IDREFS` to specify referential integrity.

   Solution:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:brk="http://somebrokerage.com/documents"
        targetNamespace="http://somebrokerage.com/documents">
  <element name="Brokerage">
    <complexType>
      <sequence>
        <element name="Broker" type="brk:brokerType"
                 minOccurs="0" maxOccurs="unbounded"/>
        <element name="Account" type="brk:accountType"
                 minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>
  <complexType name="brokerType">
    <attribute name="Id" type="ID" />
    <attribute name="Name" type="string" />
    <attribute name="Accounts" type="IDREFS" />
  </complexType>
  <complexType name="accountType">
    <attribute name="Id" type="ID" />
    <attribute name="Owner" type="string" />
    <element name="Positions" />
      <complexType>
        <sequence>
          <element name="Position">
            <complexType>
              <attribute name="stockSym" type="string" />
              <attribute name="qty" type="integer" />
            </complexType>
          </element>
        </sequence>
      </complexType>
    </element>
  </complexType>
</schema>
```

4. Consider XML documents of the form below, where actor's name and movie's title uniquely identify actors and movies, respectively.
   Write the corresponding XML schema including the key and foreign key constraints.

```
<MovieLand>
   <Actor>
      <Name>...</Name>
      <PlayedIn>
          <MovieTitle>...</MovieTitle>
                  ...
          <MovieTitle>...</MovieTitle>
      </PlayedIn>
   </Actor>
     ...
   <Actor>
      ...
   </Actor>

   <Movie>
       <Title>...</Title>
       <Actors>
          <ActorName>...</ActorName>
                  ...
          <ActorName>...</ActorName>
       </Actors>
   </Movie>
     ...
   <Movie>
     ...
   </Movie>
</MovieLand>
```

Solution: We will use only anonymous types in this solution.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:this="http://myschema.name"
        targetNamespace="http://myschema.name">

   <element name="MovieLand">
      <complexType>
         <sequence>

            <element name="Actor" minOccurs="0" maxOccurs="unbounded">
                <complexType>
                   <sequence>
                      <element name="Name" type="String"/>
                      <element name="PlayedIn">
                         <complexType>
                           <sequence>
                             <element name="MovieTitle" type="String"
                                      minOccurs="0" maxOccurs="unbounded"/>
                           </sequence>
                         </complexType>
                      </element>
                   </sequence>
                </complexType>
            </element>

            <element name="Movie" minOccurs="0" maxOccurs="unbounded">
               <complexType>
                  <sequence>
                     <element name="Title" type="String"/>
                     <element name="Actors">
                        <complexType>
                           <sequence>
                             <element name="ActorName" type="String"
                                      minOccurs="0" maxOccurs="unbounded"/>
                           </sequence>
                        </complexType>
                     </element>
                  </sequence>
               </complexType>
            </element>

         </sequence>
         <!--  include key specifications here -- next page  -->
      </complexType>
   </element>
```

```
</schema>
```

Specification of keys and foreign keys.

```
<key name="ActorKey">
   <selector xpath="Actor" />
   <field xpath="Name" />
</key>
<keyref name="ActorFK" refer="this:MovieKey">
   <selector xpath="Actor/PlayedIn"/>
   <field xpath="MovieTitle"/>
</keyref>

<key name="MovieKey">
   <selector xpath="Movie" />
   <field xpath="Title" />
</key>
<keyref name="MovieFK" refer="this:ActorKey">
   <selector xpath="Movie/Actors"/>
   <field xpath="ActorName"/>
</keyref>
```

5. Consider XML documents, below, where seller's name, buyer's name, and product name uniquely identify sellers, buyers, and products, respectively.
   Write the corresponding XML schema including the key and foreign key constraints. Seller and buyer names can include only letters and the space symbol. Product name can be an arbitrary string. Dates have the `date` type.

```
<BusinessLog>
   <Seller>
      <Name>...</Name>
      <Product name="..." price="..."/>
                  ...
      <Product name="..." price="..."/>
   </Seller>
     ...
   <Seller>
      ...
   </Seller>

   <Buyer name="...">
      <Transaction seller="..." product="..." date="..."/>
                  ...
      <Transaction seller="..." product="..." date="..."/>
   </Buyer>
     ...
   <Buyer>
     ...
   </Buyer>
</BusinessLog>
```

Solution:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://www.example.com/my-business"
        targetNamespace="http://www.example.com/my-business">

   <element name="BusinessLog">
     <complexType>
       <sequence>
          <element name="Seller" type="b:SellerType"
                   minOccurs="0" maxOccurs="unbounded"/>
          <element name="Buyer" type="b:BuyerType"
                   minOccurs="0" maxOccurs="unbounded"/>
       </sequence>
     </complexType>
     <key name="BuyerKey">
        <selector xpath="//Buyer"/>
        <field path="@name"/>
     </key>
     <key name="SellerKey">
        <selector xpath="//Seller"/>
        <field path="Name"/>
     </key>
     <key name="ProductKey">
        <selector xpath="//Seller/Product"/>
        <field path="@name"/>
     </key>
     <keyref name="SellerRef" refer="b:SellerKey">
        <selector xpath="//Transaction"/>
        <field xpath="@seller"/>
     </keyref>
     <keyref name="ProductRef" refer="b:ProductKey">
        <selector xpath="//Transaction"/>
        <field xpath="@product"/>
     </keyref>
   </element>
   <complexType name="SellerType">
      <sequence>
         <element name="Name" type="b:NameType"/>
         <element name="Product" type="b:ProductType"
                  minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
   </complexType>
   <complexType name="BuyerType">
      <sequence>
         <element name="Transaction" type="b:TransactionType"
```

```
                minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
      <attribute name="name" type="b:NameType"/>
   </complexType>
   <complexType name="ProductType">
      <attribute name="name" type="string"/>
      <attribute name="price" type="decimal"/>
   </complexType>
   <complexType name="TransactionType">
      <attribute name="seller" type="b:NameType"/>
      <attribute name="product" type="string"/>
      <attribute name="date" type="date"/>
   </complexType>

   <simpleType name="b:NameType">
      <restriction base="string">
          <pattern value="[A-Za-z ]+"/>
      </restriction>
   </simpleType>
</schema>
```

6. Formulate the following XPath queries for the document of the form

```
<Classes>
  <Class CrsCode="CS308" Semester="F1997">
    <CrsName>Software Engineering</CrsName>
    <Instructor>Adrian Jones</Instructor>
  </Class>
  <Class CrsCode="EE101" Semester="F1995">
    <CrsName>Electronic Circuits</CrsName>
    <Instructor>David Jones</Instructor>
  </Class>
  ....
</Classes>
```

(a) Find the names of all courses taught by Mary Doe in fall 1995.

Solution:

```
//CrsName[../Instructor="Mary Doe" and ../@Semester="F1995"]/text()
```

Note that we use `text()` at the end to get the text (course names themselves) as opposed to `CrsCode` element nodes.

(b) Find the set of all document nodes that correspond to the course names taught in fall 1996 or all instructors who taught `MAT123`.

Solution:

```
//CrsName[../@Semester="F1996"] | //Instructor[../@CrsCode="MAT123"]
```

(c) Find the set of all course codes taught by John Smyth in spring 1997.

Solution:

```
//Class[Instructor="John Smyth" and @Semester="S1997"]/@CrsCode
```

7. Use XSLT to transform the document in Figure 17.15 into a well-formed XML document that contains the list of `Student` elements such that each student in the list took a course in spring 1996.

   Solution:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="/">
     <StudentListS1996>
        <xsl:apply-templates/>
     </StudentListS1996>
  </xsl:template>
  <xsl:template match="//Student">
     <xsl:if test="../CrsTaken/@Semester='S1996'">
         <xsl:copy-of select="."/>
     </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

8. Write an XSLT stylesheet, which takes a document and transforms it as follows. Attributes are copied as is. Element nodes are copied with the following modifications. If a text node is a child of an element that has sub-elements, then this text node is eliminated. If a text node is a child of an element that has no e-children then this text node becomes an attribute named `text` (in this case the element has a single t-child and possibly several a-children). For instance, the document

```
<aaa bbb="1">
   foo
   <ddd eee="2">cde</ddd>
   bar
   <fff gg="3"></fff>
</aaa>
```

should be turned into

```
<aaa bbb="1">
   <ddd eee="2" text="cde"></ddd>
   <fff gg="1" text=""></fff>
</aaa>
```

Solution:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
   <xsl:template match="/">
        <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="text()">
        <!-- ignore -->
   </xsl:template>

   <xsl:template match="@*">
        <xsl:value-of select="."/>
   </xsl:template>

   <!-- matches elements with e-children -->
   <xsl:template match="*[*]">
        <xsl:copy>
              <xsl:apply-templates match="@*"/>
<!-- Note: apply-templates also handles text nodes -->
              <xsl:apply-templates/>
        </xsl:copy>
   </xsl:template>

   <!-- lesser precedence: matches the remaining elements -->
   <xsl:template match="*">
        <xsl:copy>
              <xsl:apply-templates match="@*"/>
              <xsl:attribute name="text">
                   <xsl:value-of select="text()"/>
              </xsl:attribute>
              <!-- No need to apply-templates for other notes:
                   the current element has no e-children and text
                   nodes have already been taken care of -->
        </xsl:copy>
   </xsl:template>

</xsl:stylesheet>
```

9. Write an XSLT stylesheet that traverses the document tree and, ignoring attributes, copies the elements and *doubles* the text nodes. For instance, <foo a="1">the<best/>bar</foo> would be converted into <foo>thethe<best/> barbar</foo>.

   Solution:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
  <xsl:template match="/|*">
     <xsl:copy>
        <xsl:apply-templates/>
     </xsl:copy>
  </xsl:template>
  <xsl:template match="text()">
     <xsl:value-of select=".">
     <xsl:value-of select=".">
  </xsl:template>
</xsl:stylesheet>
```

10. Write an XSLT stylesheet, which takes a document and transforms it as follows. The element nodes and attribute nodes are copied as is. However, the text nodes that are children of an element node are deleted and replaced with an attribute called `text`. The value of the `text` attribute is a string that is a concatenation of the strings represented by the deleted text nodes. For instance,

```
<aaa bbb="1">
   abc <dddd eee="2">cde</dddd> fgh
   <fff gg="3" />
</aaa>
```

should become

```
<aaa bbb="1" text="abcfgh">
   <dddd eee="2" text="cde"></dddd>
   <fff gg="1" text="" />
</aaa>
```

Hint: An attribute `text` can be created using the XSLT instruction

```
<xsl:attribute name = "text">
   <!-- Content -->
</xsl:attribute>
```

similarly to the way new elements are created in XSLT using the `xsl:element` instruction. A concatenation of all t-children of an element can be obtained with the XPath selector `text()`.

Solution: The XSLT transformation is as follows:

```
<xsl:template match="text()">
  <!-- ignore -->
</xsl:template>

<xsl:template match="*">
  <xsl:copy>
    <xsl:attribute name="text()">
      <xsl:value-of select="concat(./text())"/>
    </xsl:attribute>

    <xsl:apply-templates/>
    <xsl:apply-templates select="@*"/>

  </xsl:copy>
</xsl:template>

<xsl:template match="@*">
  <xsl:copy-of select="."/>
</xsl:template>
```

11. Consider a document of the form

```
<authors>
  <author name="..." >
     <book title="..." year="..." />
     <book title="..." year="..." />
     <book title="..." year="..." />
      ...  ...  ...
  </author>
  <author name="..." >
     <book title="..." year="..." />
     <book title="..." year="..." />
     <book title="..." year="..." />
      ...  ...  ...
  </author>
  ...  ...  ...
</authors>
```

Use XQuery to find the years in which the most books were published. For each such year, list the book titles that were published in that year. The output should look like this:

```
<bestYear>
     <year bookCount="..." year="...">
        <book title="..."/>
        <book title="..."/>
        ... ... ...
     </year>
     ... ... ...
</bestYear>
```

Solution:

```
DECLARE FUNCTION local:booksByYearWithCount() AS element()* {
   FOR  $y IN fn:distinct-values(fn:doc("input.xml")//@year)
   RETURN <year year=$y
                bookCount={fn:count(fn:distinct-values(
                                fn:doc("input.xml")//book[@year=$y]
                                    ))} >
            {
               FOR $b IN fn:distinct-values(
                             fn:doc("input.xml")//book[@year=$y])
               RETURN <book title={$b/@title} />
            }
          </year>
}

<bestYear>
  {
    LET $max := fn:max(local:booksByYearWithCount()/@bookCount)
    FOR $b IN local:booksByYearWithCount()
    WHERE $b/@bookCount = $max
    RETURN $b
  }
</bestYear>
```

12. Consider the following document structure for the transcript records:

```
<Transcript>
  <tuple>
    <StudId value="..."/> <CrsCode value="..."/>
    <Semester value="..."/> <Grade value="..."/>
  </tuple>
  ... ... ...
</Transcript>
```

Use this structure to formulate the following queries in XQuery:

(a) List all classes (identified by a course code and semester) where every student received a B or higher.

Solution:

```
DECLARE FUNCTION local:classes() AS element()*  {
  FOR $c  IN  fn:doc("http://xyz.edu/transcript.xml")//tuple
  RETURN
      <class> $s//CrsCode, $c//Semester </class>
}

<EasyClasses>
(
  FOR $c  IN fn:distinct-values(local:classes())
  LET $grades :=
           fn:doc("http://xyz.edu/transcript.xml")
                 //tuple[CrsCode/@value=$c/CrsCode/@value
                         and Semester/@value=$c/Semester/@value]
                      /Grade/@value
  WHERE   EVERY $g  IN $grades
            SATISFIES local:numericGrade($g) >= local:numericGrade("B")
  RETURN $c
)
</EasyClasses>
```

(b) List all students who never received less than a B.

Solution:

```
<GoodStudents>
(
 FOR $sid  IN
    fn:distinct-values(fn:doc("http://xyz.edu/transcript.xml")
                                  //tuple/StudId/@value)),
    $s  IN fn:doc("http://xyz.edu/student.xml")
                                  //tuple[Id/@value=$sid]
```

```
  LET $grades :=
         fn:doc("http://xyz.edu/transcript.xml")
               //tuple[StudId/@value=$sid]/Grade/@value
  WHERE   EVERY $g  IN $grades
            SATISFIES local:numericGrade($g) >= local:numericGrade("B")
  RETURN $s
)
</GoodStudents>
```

13. Write an XQuery function that traverses a document and computes the maximum branching factor of the document tree, i.e., the maximal number of children (text or element nodes) of any element in the document.

Solution:

```
NAMESPACE  xsd = "http://www.w3.org/2001/XMLSchema"
DECLARE FUNCTION local:maxBranching($n)  RETURNS xsd:integer
  {
        LET $children := $n/node()
        LET $maxChildBranching :=
                    fn:max( FOR $m  IN $children
                            RETURN local:maxBranching($m))
        LET $childCount := fn:count($children)
        RETURN
              IF $childCount >= $maxChildBranching
              THEN $childCount
              ELSE $maxChildBranching
  }
```

Recall that the XPath function `node()` matches every *e*- or *t*-node.

14. Consider a document of the form

```
<stores>
  <store name="..." city="...">
     <item name="..." price="...">
     <item name="..." price="...">
     <item name="..." price="...">
      ...  ...  ...
  </store>
  <store name="..." city="...">
     <item name="..." price="...">
     <item name="..." price="...">
     <item name="..." price="...">
      ...  ...  ...
  </store>
  ...  ...  ...
</stores>
```

Assume that name and city uniquely determine a store and within each store there is at most one item with a given name. Write the following query using XQuery:

> *Find the stores with the* **lowest** *average price of items among all the stores in New York City.*

Return the names of all such stores.

Solution:

```
<Result>
  for $s in fn:doc("...")/store[@city="NYC"]
  where fn:avg($s//@price) =
              min( for $s1 in fn:doc("...")/store[@city="NYC"]
                   return fn:avg($s1//@price) )
  return <storeName> { $s/name } </storeName>
</Result>
```

15. Consider a relation with the following schema: STORE(Name,Items), where Name is a character string and Items is a column that stores data using the XML data type of SQL/XML. The documents in the Items column have the following form:

```
<items>
     <item name="..." price="...">
     <item name="..." price="...">
     <item name="..." price="...">
</items>
```

Use SQL/XML to answer the query:

Find the stores with the **lowest** *average price of the items being sold.*

Solution 1:

```
CREATE VIEW AVERAGES(Name,Average) AS
   SELECT S.Name,
          XMLQUERY(
             'RETURN avg(RETURN $items//@price)'
             RETURNING CONTENT
             PASSING BY REF S.Items AS items
          ).getNumberVal()
   FROM STORE S

SELECT A.Name
FROM AVERAGES A
WHERE A.Average = ( SELECT min(A.Average)
                    FROM AVERAGES A )
```

The only non-obvious feature here is the `getNumberVal()` function. When we specify RETURNING CONTENT in the XMLQUERY statement, the construct returns the result as an XML fragment represented as a string. In our case, this fragment is a single node of type integer. The built-in function `getNumberVal()` takes this XML node and extracts the number value to make it appropriate for use in SQL.

Solution 2:

```
SELECT S.Name,
FROM   STORE S
       -- This creates a list of all <items> elements
       -- in the Items column. S2 is now bound to that list
       ( SELECT  XMLAGG(S1.Items)  AS List  FROM  STORE S1)  AS S2
WHERE
   XMLEXISTS( XMLQUERY(
     'LET $Avg := fn:avg($I//price)
      WHERE $Avg =
               fn:min(FOR $A IN $I1/items RETURN fn:avg($A//price))
      RETURN $I'
    PASSING BY REF S.Items  AS I,
                   S2.List  AS I1
   ))
```

This query looks simple, but it requires a non-obvious hack. We cannot have SQL statements inside XQuery statements, so we had to create a list of the form <items>...</items> ... <items>...</items> using SQL. Then we bind this list to an SQL variable and pass that variable to XQuery.

Problems for Chapter 16: Distributed Databases

1. Show that the semijoin operation is not commutative, that is, $\mathbf{T}_1 \ltimes_{join-condition} \mathbf{T}_2$ is not the same as $\mathbf{T}_2 \ltimes_{join-condition} \mathbf{T}_1$.

   Solution:

   Informally, the semijoin of $\mathbf{T}_1$ and $\mathbf{T}_2$, based on a join condition, consists of the tuples of $\mathbf{T}_1$ that participate in the join with $\mathbf{T}_2$. Clearly the semijoin is not commutative, because that would imply that the semijoin of $\mathbf{T}_2$ and $\mathbf{T}_1$, based on the same join condition, consists of the tuples of $\mathbf{T}_2$ that participate in the join and that the tuples of $\mathbf{T}_1$ that participate in the join are the same as the tuples of $\mathbf{T}_2$ that participate in the join.

   More formally, $\mathbf{T}_1 \ltimes_{join-condition} \mathbf{T}_2$ is defined to be the projection over the columns of $\mathbf{T}_1$ of the join of $\mathbf{T}_1$ and $\mathbf{T}_2$:

   $$\pi_{attributes(\mathbf{T}_1)} (\mathbf{T}_1 \bowtie_{join\text{-}condition} \mathbf{T}_2)$$

   Again the projection over the attributes of $\mathbf{T}_1$ is different than the projection over the attributes of $\mathbf{T}_2$

2. Design a query plan for the following distributed query: An application at site B wants to compute a join STUDENT $\bowtie_{\texttt{Id=StudId}}$ TRANSCRIPT, where STUDENT(`Id,Major`) is t site B and TRANSCRIPT(`StudId,CrsCode`) is at site C. The result should be returned to B. Assume that semijoin is not used. Also assume that

   - The various lengths are:
     - `Id` and `StudId` are 8 bytes long;
     - `Major` is 3 bytes long;
     - `CrsCode` is 6 bytes long.
   - STUDENT has 15,000 tuples.
   - 6,000 students are registered for at least one course. On the average, each student is registered for 5 courses.

   Solution:

   TRANSCRIPT has 6,000*5=30,000 tuples, each 14 (8+6) bytes long. Each STUDENT tuple has 11 (8+3) bytes. The join has 8+6+3=17 bytes in each tuple.

   1. If we send STUDENT to site C, compute the join there, and then send the result to site B, we have to send 675,000 bytes ($= 15,000 * 11 + 30,000 * 17$).

   2. If we send TRANSCRIPT to site B and compute the join there, we have to send 420,000 bytes ($= 30,000 * 14$).

   Thus, alternative 2 is better.

3. Consider the following schema:

   Movie(Title, Director, ReleaseYear)
   ShownIn(Title, MovieTheater)

   $\pi_{\texttt{MovieTheater}}(\sigma_{\texttt{ReleaseYear='2000'}}(\text{ShownIn} \bowtie_{\texttt{Title}} \text{Movie}))$

   Assume that values for each attribute are 20 bytes long; Movie has 5,000 tuples, where on the average each year saw 50 new movies. Each movie is shown in 50 theaters on average.

   Rewrite this query using the semijoin operators $\ltimes$ *or* $\rtimes$ *and show the optimal (in terms of network traffic)* plan. Assume that the relation ShownIn is at site A, Movie at site B, and the query is issued at site C.

Solution:

Since each movie is shown in 50 theatres on average, ShownIn must have about 250,000 tuples. Moving either relation in its entirety is likely to be prohibitively costly.

It is obvious that we first need to compute the selection:

$$\sigma_{\texttt{ReleaseYear='2000'}}(\text{Movie})$$

which leaves about 50 tuples.

The best plan is to use left semijoin because the output involves only the MovieTheater attribute, which comes from relation ShownIn, which is at the same site as the one where the semijoin will be finally computed.

So, the best plan is described by the following expression:

$$\pi_{\texttt{MovieTheater}}(\text{ShownIn} \ltimes_{\texttt{Title}} \sigma_{\texttt{ReleaseYear='2000'}}(\text{Movie}))$$

The plan looks as follows:

(a) Move $\pi_{\texttt{Title}}(\sigma_{\texttt{ReleaseYear='2000'}}(\text{Movie}))$ from site B to site A. Cost: 50*20=1000 bytes.

(b) Do the join part of the semijoin at site A. The join contains 50 titles * 50 theatres = 2,500 tuples.

(c) Move the projection $\pi_{\texttt{MovieTheater}}$ of the join to site C. In the worst case, all movie theatres in the join are distinct, so in the worst case we will move 2,500 items at cost 2,500 * 20 = 50,000 bytes.

4. Estimate the cost of computing $\sigma_{\texttt{Major='CS'}}(\textsc{Student}) \bowtie_{\texttt{Id=StudId}} \textsc{Transcript}$ using the semijoin strategy. Use the sizes of the STUDENT and TRANSCRIPT relations and of their attributes as follows.

   - The attribute lengths:
     - `Id` and `StudId` are 8 bytes long;
     - `Major` is 3 bytes long;
     - `CrsCode` is 6 bytes long.
   - STUDENT has 15,000 tuples.
   - 6,000 students are registered for at least one course. On the average, each student is registered for 5 courses.

   In addition, assume that 10% of the students major in `CS`.

   Compare this with solutions that do not not use the semijoin.

   Solution:

   Send the projection of $\sigma_{\texttt{Major='CS'}}(\textsc{Student})$ on `Id` to site C: 8*1,500 = 12,000 bytes. Among these, the number of registered students can be estimated as 1,500*6,000/15,000 = 600. Therefore, the join at site C will have 600*5=3,000 tuples. Send them to B at cost 3,000 * (8+6=14) = 42,000. The total cost is 42,000+12,000 = 54,000.

   If we use Alternative 2 in the previous problem, then the cost is still 420,000. If we use Alternative 1, we can send only the CS majors to site C: 1,500 * 11 = 16,500. Since the join was previously estimated to have 3,000 tuples, sending them to B will cost 3,000 * 14 = 42,000 bytes. The total therefore is 42,000+16,500=58,500. Therefore, the semijoin-based plan is better.

5. Consider the following query:

$$\pi_{\texttt{CustId,Price}}(\sigma_{\texttt{Category='Diapers'}}(\textsc{Transaction} \bowtie_{\texttt{ItemId=Id}} \textsc{Item}))$$

Assume that TRANSACTION has the attributes `CustId`, `ItemId`, and `Date`. The ITEM relation has the attributes `Id`, `Category`, and `Price`, where `Id` is a key.

(a) Rewrite this query using the semijoin operator $\ltimes$ or $\rtimes$.

(b) Find an optimal query evaluation plan with respect to the communication cost.

Questions (a) and (b) are independent of each other (the answer in (a) does not need to be optimal with respect to the cost).

For question (b), assume that the query was issued at site A; the TRANSACTION relation is at site B with 10,000 tuples; and the ITEM relation is at site C with 1,000 tuples. Further assume that each tuple is 12 bytes long and each value of an attribute requires 4 bytes. There are 10 items in a category on the average. The selectivity factor of any particular item (i.e., the probability that it appears in any given tuple in TRANSACTION) is 0.001.

Solution:

(a) For example, $\pi_{\texttt{CustId,Price}}((\textsc{Transaction} \ltimes \sigma_{\texttt{Category='Diapers'}}(\textsc{Item})) \bowtie \sigma_{\texttt{Category='Diapers'}}(\textsc{Item}))$

(b) i. Compute $\pi_{\texttt{Id,Price}}(\sigma_{\texttt{Category='Diapers'}}(\textsc{Item}))$ which has the size 10 tuples * 8 bytes = 80 bytes.
Send the result to site B.

ii. Perform a join at site B.
On the average, every item occurs in 10000 * 0.001 = 10 tuples, so the join has 100 tuples.
Projecting the result on `CustId` and `Price` makes each tuple 8 bytes long. Therefore, the size of the result is 100*8=800 bytes. This result is sent to site A. Total cost: 880.

Note that the semijoin is not optimal for the following reasons. First, computing the right semijoin $\textsc{Transaction} \rtimes \sigma_{\texttt{Category='Diapers'}}(\textsc{Item})$ is expensive. Projecting TRANSACTION on `ItemId` can have as many as 1,000 item Ids, each 4 bytes long. Sending this to site C costs 4,000, which is way more than 880.

Computing the other semijoin, $\textsc{Transaction} \ltimes \sigma_{\texttt{Category='Diapers'}}(\textsc{Item})$ means sending the 10 item Ids contained in $\sigma_{\texttt{Category='Diapers'}}(\textsc{Item})$ from C to B at the cost of 40. The semijoin result computed at B will have 100 tuples, as before. To complete the query, we have to send that result to either C or A at the cost of 800. In the first case, the result is computed at C and then sent to A. Since the result has 100 2-attribute tuples, it means that the cost of sending that result to A is 800, and the total cost will be 800+800+40=1640. In the second case, $\pi_{\texttt{Id,Price}}(\sigma_{\texttt{Category='Diapers'}}(\textsc{Item}))$ has to be sent to A at the cost of 80. So, the total cost will be 920.

6. Consider the following schema:

   PART(Id, Name, Description)
   WAREHOUSE(Location, PartId, Quantity)

   The keys in each relation are underlined. Assume that Id and PartId data items are each 6 bytes long, Name and Location are 20 bytes long each, Quantity is 4 bytes long, and Description is 100 bytes long.

   Suppose that PART is at site **A** and the WAREHOUSE relation is horizontally partitioned by location, such that for each value of the Location attribute the relation $\sigma_{Location=c}$(WAREHOUSE) is at its own site $\mathbf{B}_c$.

   Consider the following query that is posed by a user at site **C**, which is different from **A** and from all the $\mathbf{B}_c$'s.

```
SELECT P.Name
FROM PART P
WHERE 1000 < (SELECT sum(W.Quantity)
              FROM WAREHOUSE W
              WHERE P.Id = W.PartId)
```

   How should the data be replicated at sites **A** and/or the various $\mathbf{B}_c$ sites so that the following criteria will be optimized:

   - The above query will be evaluated with the least amount of network traffic.
   - Among the options that minimize the first criterion, the amount of storage spent on replication is minimal.

   Explain your answer.

   Solution:

   Since we need to count the quantities over all locations for any given part number, traffic will be minimized if $\pi_{\texttt{PartId,Quantity}}$(WAREHOUSE) is replicated at site **A**. Then the entire query can be computed at site **A** at no transmission cost and the result is sent to site **C**.

   This minimizes transmission because the cost of transmission is reduced to just sending the result to **C** (this cost cannot be any lower). It minimizes storage because there is no alternative way to minimize transmission costs.

Problems for Chapter 17: OLAP and Data Mining

1. Design SQL queries for the supermarket example of Section 17.2 in Chapter 17 that will return the information needed to make a table of the form below, where grouping happens by months and states.

		State		
SUM(Sales_Amt)		S1	S2	S3	Total
	M1	2	5	4	11
Month	M2	4	2	7	13
	M3	8	1	3	12
	M4	5	9	2	16
Total		19	17	16	52

   (a) Use CUBE or ROLLUP operators.
   Solution:

```
SELECT M.State, T.Month, SUM(S.Sales_Amt)
FROM   Sales S, Market M, Time T
WHERE  S.Market_Id = M.Market_Id AND S.Time_Id = T.Time_Id
GROUP BY CUBE (M.State, T.Month)
```

   (b) Do not use CUBE or ROLLUP operators.
   Solution:

```
SELECT  M.State, T.Month, SUM(S.Sales_Amt)
FROM    Sales S, Market M, Time T
WHERE   S.Market_Id = M.Market_Id AND S.Time_Id = T.Time_Id
GROUP BY  M.State, T.Month

SELECT  M.State, SUM(S.Sales_Amt)
FROM    Sales S, Market M
WHERE   S.Market_Id = M.Market_Id
GROUP BY  M.State


SELECT  T.Month, SUM(S.Sales_Amt)
FROM    Sales S, Time T
WHERE   S.Time_Id = T.Time_Id
GROUP BY  T.Month


SELECT   SUM(S.Sales_Amt)
FROM    Sales S
```

2. Consider the following two-dimensional cube:

SALES	Time t1	t2	t3	t4
Location NY	3	5	2	3
Location PA	4	5	2	2
Location OR	1	3	2	2
Location WA	5	4	2	1

where Location and Time are dimensions represented by the following tables:

TIME	TimeId	Month
	t1	January
	t2	January
	t3	February
	t4	February

LOCATION	State	Region
	NY	NE
	PA	NE
	OR	NW
	WA	NW

Assume that the cube represents a relation with the following schema:

SALES(Time,Location,Amount)

Consider the following OLAP query:

```
SELECT T.Month, L.Region, SUM(S.Amount)
FROM SALES S, TIME T, LOCATION L
WHERE S.Time = T.TimeId AND S.Location = L.State
GROUP BY ROLLUP(T.Month, L.Region)
```

(a) Show the actual SQL queries that will be posed as a result of this single rollup.

(b) Show the cubes (of 2, 1, and 0 dimensions) produced by each of these queries.

(c) Show how the lower-dimension cubes are computed from higher-dimension cubes.

Solution:

(a) The queries are:

```
SELECT T.Month, L.Region, SUM(S.Amount)
FROM SALES S, TIME T, LOCATION L
WHERE S.Time = T.TimeId AND S.Location = L.State
GROUP BY T.Month, L.Region
```

```
SELECT T.Month, SUM(S.Amount)
FROM SALES S, TIME T, LOCATION L
WHERE S.Time = T.TimeId AND S.Location = L.State
GROUP BY T.Month
```

```
SELECT SUM(S.Amount)
FROM SALES S, TIME T, LOCATION L
WHERE S.Time = T.TimeId AND S.Location = L.State
```

(b) The cubes produced by these queries are:

```
Sales | January   February
------+-------------------
  NE  |   17          9
  NW  |   13          7
```

```
Sales | January   February
------+-------------------
      |   30          16
```

```
Sales |
------+------
      |  46
```

(c) The second cube was produced from the first one by adding rows. The third cube was obtained from the second one by adding columns.

3. This problem is similar to Problem 2, but it uses CUBE instead of ROLLUP. Consider the following two-dimensional cube:

SALES		Time			
		t1	t2	t3	t4
Location	NY	3	5	2	3
	PA	4	5	2	2
	OR	1	3	2	2
	WA	5	4	2	1

where Location and Time are dimensions represented by the following tables:

TIME	TimeId	Month
	t1	January
	t2	January
	t3	February
	t4	February

LOCATION	State	Region
	NY	NE
	PA	NE
	OR	NW
	WA	NW

Assume that the cube represents a relation with the following schema:

SALES(Time,Location,Amount)

Consider the following OLAP query:

```
SELECT T.Month, L.Region, SUM(S.Amount)
FROM SALES S, TIME T, LOCATION L
WHERE S.Time = T.TimeId AND S.Location = L.State
GROUP BY CUBE(T.Month, L.Region)
```

(a) Show the actual SQL queries that will be posed as a result of this cube operation.

(b) Show the cubes (of 2, 1, and 0 dimensions) produced by each of these queries.

(c) Show how the lower-dimension cubes are computed from higher-dimension cubes.

Solution:

(a) The SQL queries corresponding to the above CUBE statement are:

(i)

```
SELECT T.Month, L.Region, SUM(S.Amount)
FROM SALES S, TIME T, LOCATION L
WHERE S.Time = T.TimeId AND S.Location = L.State
GROUP BY T.Month, L.Region
```

(ii)

```
SELECT T.Month, SUM(S.Amount)
FROM SALES S, TIME T, LOCATION L
WHERE S.Time = T.TimeId AND S.Location = L.State
GROUP BY T.Month
```

(iii)

```
SELECT L.Region, SUM(S.Amount)
FROM SALES S, TIME T, LOCATION L
WHERE S.Time = T.TimeId AND S.Location = L.State
GROUP BY L.Region
```

(iv)

```
SELECT SUM(S.Amount)
FROM SALES S, TIME T, LOCATION L
WHERE S.Time = T.TimeId AND S.Location = L.State
```

(b) The SQL statement (i) computes the following 2-dimensional cube:

```
S1          January   February
          |----------------------
      NE  |    17         9
          |
      NW  |    13         7
```

The statement (ii) computes this 1-dimensional cube:

```
S2          January   February
          |---------------------
               30        16
```

(iii) computes this 1-dimensional cube:

```
S3
          |
      NE  |    26
          |
      NW  |    20
```

(iv) computes this 0-dimensional cube:

```
 S4
       46
```

(c) Cube S2 is obtained from cube S1 by summing up the elements of each column. S3 is obtained from S1 by summing up the values in each row. Finally, S4 can be obtained from either S2 or S3 by summing up the values in the single row/column, respectively.

4. Perform the a priori algorithm on the following table to determine all two-item associations with support of at least 0.3 and confidence of at least 0.5.

PURCHASES	Transaction_id	Product
	001	diapers
	001	beer
	001	popcorn
	001	bread
	002	diapers
	002	cheese
	002	soda
	002	beer
	002	juice
	003	diapers
	003	cold cuts
	003	cookies
	003	napkins
	004	cereal
	004	beer
	004	cold cuts

Solution: The support for the various items is as follows:

Item	Support
diapers	3/4 = 0.75
beer	3/4 = 0.75
popcorn	1/4 = 0.25
bread	1/4 = 0.25
cheese	1/4 = 0.25
soda	1/4 = 0.25
juice	1/4 = 0.25
cold cuts	2/4 = 0.5
cookies	1/4 = 0.25
napkins	1/4 = 0.25
cereal	1/4 = 0.25

It follows that only the associations among diapers, beer, and cold cuts can *possibly* have support of 0.4 or more. We need to calculate the actual support, to be sure:

Item-pair	Support
diapers,beer	2/4 = 0.5
diapers,cold cuts	1/4 = 0.25
beer,cold cuts	1/4 = 0.25

Therefore, the only associations with support of at least 0.4 can occur between beer and diapers. Both of the following associations have confidence of 0.5:

Purchase_diapers => Purchase_beer
Purchase_beer => Purchase_diapers

5. Consider the table

TransactionId	Product
1	milk
1	beer
1	diapers
2	beer
2	diapers
3	duck
3	yogurt
3	milk
4	milk
4	beer
4	paper
5	diapers
5	beer
6	diapers
6	duck
6	paper

Perform the a priori algorithm to find two-item associations with support of at least 40%. Determine the confidence of each association that you found.

Solution: The support for the individual items are:

Item	Support
milk	3/6 = 0.5
beer	4/6 = 0.66
diapers	4/6 = 0.66
duck	2/6 = 0.33
yogurt	1/6 = 0.16
paper	2/6 = 0.33

Since we are interested for associations with support of at least 40%, each item in the association must have support of at least 40%. We have only three such items: milk, beer, and diapers, so only 3 potential associations need be considered:

Item-pair	Support
milk, beer	2/6 = 0.33
beer, diapers	3/6 = 0.5
milk, diapers	1/6 = 0.16

So, only the associations that involve diapers and beer have support more than 40%. It remains to compute the confidence of two possible associations:

Association	Confidence
diapers => beer	3/4 = 0.25
beer => diapers	3/4 = 0.25

6. Consider the following table, which records past performance of students in a course. The last column (Failed) represents the outcome, and columns 2, 3, 4 are the attributes used in making the decision.

Student	Prerequisites	AvgGrade	FailedBefore	Failed
1	yes	3	no	no
2	yes	2	no	no
3	no	4	no	no
4	yes	1	yes	yes
5	no	2	no	no
6	yes	2	yes	yes
7	yes	3	yes	no
8	no	3	yes	yes
9	yes	1	no	yes
10	yes	4	yes	no

Create a decision tree for predicting the outcome of the course using the ID3 algorithm. Instead of the entropy, which is hard to compute by hand, use the Gini index to measure the randomness of the outcomes in the intermediate tables during the run of the ID3 algorithm:

$$Gini(T) = 1 - (p_{yes}^2 + p_{no}^2)$$

where p_{yes} and p_{no} are the fractions of items in table T that have outcome *yes* and *no*, respectively.

Hint: You only need to do calculations to choose the root of the tree. The choice of the attributes for the second level of the tree will become apparent after you make the computations for the root.

Solution:

Compute the **Gini index of the entire table**:
outcome yes: 4 tuples
outcome no: 6 tuples
$Gini = 1 - (4/10)^2 - (6/10)^2 = 100/100 - 52/100 = 48/100$.

Now we compute the **information gain of** `Prerequisites`. The yes subtable: 7 tuples; 3 – outcome=yes, 4 outcome=no.
The no subtable: 3 tuples; 1 outcome=yes, 2 outcome=no.
Gini of the yes subtable: $1 - (4/7)^2 - (3/7)^2 = 49/49 - 25/49 = 24/49$.
Gini of the no subtable: $1 - (1/3)^2 - (2/3)^2 = 9/9 - 5/9 = 4/9$.

Therefore, the information gain of choosing `Prerequisites` is:
$48/100 - ((24/49) * 7/10 + (4/9) * 3/10) = 48/100 - (24/70 + 4/30) = 48/100 - 100/210 = 98/200 - 100/2 * 210 < .24$.

Now, compute the **information gain of** `FailedBefore`.

The yes subtable: 5 tuples; 3 – outcome=yes, 2 – outcome=no.
The no subtable: 5 tuples; 1 – outcome=yes, 4 – outcome=no.
Gini index of the yes subtable: $1 - (3/5)^2 - (2/5)^2 = 12/25$.
Gini index for the no subtable: $1 - (1/5)^2 - (4/5)^2 = 8/25$.
Information gain: $48/100 - ((12/25) * 1/2 + (8/25) * 1/2) = 48/100 - 20/50 = 8/100 = 0.08$.

Hence, the information gain of `FailedBefore` is lower than `Prerequisites`.

Now, compute the **information gain of** `AvgGrade`.

There are 4 possibilities.
The `AvgGrade` = 1 subtable: 2 tuples; 2 – outcome=yes, 0 – outcome=no.
The `AvgGrade` = 2 subtable: 3 tuples; 1 – outcome=yes, 2 – outcome=no.
The `AvgGrade` = 3 subtable: 3 tuples; 1 – outcome=yes, 2 – outcome=no.
The `AvgGrade` = 4 subtable: 2 tuples; 0 – outcome=yes, 2 – outcome=no.
Gini index for AvgGrade = 1 and 4 are 0: $1 - (2/2)^2 - (0/2)^2$.
Gini index for AvgGrade = 2 and 3: $1 - (1/3)^2 - (2/3)^2 = 4/9$.
Information gain of `AvgGrade` is: $48/100 - (0 * 2/10 + (4/9) * 3/10 + (4/9) * 3/10 + 0 * 2/10) = 48/100 - 8/30 > 0.24$.

Hence, `AvgGrade` gives the highest information gain. The tree can now be constructed as follows. Put `AvgGrade` at the root. If `AvgGrade` = 1 or 4, then the resulting subtrees are homogeneous: their outcomes are yes and no, respectively. It is easy to see that when `AvgGrade` = 2, the `FailedBefore` attribute splits the corresponding subtable into homogeneous tables. Likewise, when `AvgGrade` = 3, the attribute `Prerequisites` splits the corresponding subtable homogeneously. So, the rest of the tree can be constructed without much computation.

The resulting decision tree looks as follows:

```
AvgGrade -- 1 --> outcome=yes
  |
  +-------- 2 ---- FailedBefore -- yes --> outcome=yes
  |                      |
  |                      +-------- no  --> outcome=no
  |
  +---------3 ---- Prerequisites -- yes --> outcome=no
  |                      |
  |                      +--------- no  --> outcome=yes
  +-------- 4 ---- outcome=no
```

7. Consider the following table:

Company	Sales
1	30
2	100
3	120
4	60
5	20
6	70
7	25
8	75
9	110

Use the K-means algorithm to cluster the above data. Consider different numbers of clusters: 2, 3, and 4. Which number of clusters is the most meaningful here? Explain your answer.

Solution:

The case of 2 clusters: Choose random centers: 10 and 70. Then the algorithm works like this.

1st pass: Cluster #1 – 30, 20, 25 – new center 25.
Cluster #2 – 110, 75, 70, 60, 120, 100 – new center 89.

No changes at the 2nd pass.

The case of 3 clusters: Choose random centers: 10, 70, 110. Then the algorithm works like this.

1st pass: Cluster #1 – 30, 20, 25 – new center 25.
Cluster #2 – 75, 70, 60 – new center 68.
Cluster #3 – 110, 120, 100 – new center 110.

No changes at the 2nd pass.

The case of 4 clusters: Choose random centers: 10, 50, 90, 110. Then:

1st pass: Cluster #1 – 30, 20, 25 – new center 25.
Cluster #2 – 70, 60 – new center 65.
Cluster #3 – 75 – new center 75.
Cluster #3 – 100, 110, 120 – new center 110.

2nd pass: 70 moves from cluster 2 to 3.
Cluster #2 – 60 – new center 60.
Cluster #3 – 70, 75 – new center 72.5.

No changes in the 3d pass.

Note: In the first pass, item 75 could go in cluster 2 and 3, since it is equidistant from the two centers. Likewise, 100 could go to cluster 3 or 4.

The 3-cluster classification is most reasonable, since the average number of members in each cluster is most balanced and the average distance to cluster centers is adequately small.

8. Consider the following table, which records information about past customers' transactions. The last column (`WillBuyAgain`) represents the outcome, and columns 2, 3 are the attributes used in making the decision.

Customer	BoughtBefore	PaymentMethod	WillBuyAgain
1	yes	cash	yes
2	yes	check	yes
3	no	CC	yes
4	yes	cash	no
5	no	check	yes
6	yes	CC	no
7	yes	CC	yes
8	no	cash	no
9	yes	cash	no
10	yes	check	yes

Create a decision tree for predicting the outcome using the ID3 algorithm. Instead of the entropy, which is hard to compute without a calculator, use the following formula to measure randomness of the outcomes in the intermediate tables during the run of the ID3 algorithm:

$$m(T) = 1 - (p_{yes} - p_{no})^2$$

where p_{yes} and p_{no} are the fractions of the yes/no items that have outcomes *yes* and *no*, respectively.

Solution:

In the initial table T, $p_{yes} = 6/10$ and $p_{no} = 4/10$. So, the measure of randomness, $m(T) = 1 - 1/25 = 24/25$.

Let us try to use the attribute BoughtBefore as the root. This splits T into two tables, T_1 (*BoughtBefore*=yes) and T_2 ((*BoughtBefore*=no).

The measures of randomness are as follows: m(T_1) = $1 - 1/49 = 48/49$. m(T_2) = 1 - 1/9 = 8/9. Therefore, the information gain is: infogain(*BoughtBefore*) = 24/25 - (7/10*48/49+3/10*8/9) = 0.96 - 0.95 = 0.01.

Let us try to use *PaymentMethod*. This splits T into T'_1 (*PaymentMethod*=cash), T'_2 (*PaymentMethod*= check, and T'_3 (*PaymentMethod*=CC).

The randomness factors are as follows: m(T'_1) = 1-(1/4-3/4)^2 = 1-1/4=3/4. m(T'_2) = 1-1=0. m(T'_3) = 1 - (1/3-2/3)^2 = 1-1/9 = 8/9. The information gain is infogain(*PaymentMethod*) = 24/25 - (4/10*3/4+3/10*0+3/10*8/9) = 0.96 - 0.57 = 0.39.

Since infogain(*PaymentMethod*) > infogain(*BoughtBefore*), we use *PaymentMethod* as the root. Then we try to further refine the tree by examining the effect of splitting the tables T'_1, T'_2, and T'_3 according to attribute *BoughtBefore*.

Using *BoughtBefore* on table T'_1 (which corresponds to the cash-branch of the tree) yields two subtables: T'_{11} (*BoughtBefore*=yes) and T'_{12} (*BoughtBefore*=no). T'_{11} has items with different outcomes, so this branch will misclassify some cases. Table T'_{12} has only items with outcome "yes", so this branch is not expected to produce classification errors.

Table T'_2 is homogeneous, so we do not need to expand this branch further.

Using *BoughtBefore* on table T'_3 (which corresponds to the CC-branch of the tree) yields two subtables: T'_{31} (*BoughtBefore*=yes) and T'_{32} (*BoughtBefore*=no). T'_{31} has items with different outcomes, so this branch will misclassify some cases. Table T'_{32} has only items with outcome "yes", so this branch is not expected to produce classification errors.

Therefore, the decision tree looks like this:

```
 PaymentMethod
  |     |    |
 cash  check CC
  |     |    |
  |     |    |
  |     |   BoughtBefore
  |     |    |
  |     |    +-- no --- WillBuyAgain=yes
  |     |    |
  |     |    +-- yes -- WillBuyAgain=yes/no (50-50% chance)
  |     |
  |   WillBuyAgain=yes
  |
BoughtBefore
  |
  +--- no --- WillBuyAgain=no
  |
  +--- yes --- WillBuyAgain=yes/no (33.3-66.6% chance)
```

9. Consider the following table:

Company	Sales
1	40
2	105
3	120
4	25
5	70
6	25
7	75
8	85
9	30

Show the workings of the hierarchical algorithm on the above data. Draw the dendogram produced by your run of that algorithm.

Solution:

The steps are illustrated in Figure 2. The numbers at vertexes denote the centers of the clusters that are formed by the items in the leaves below those vertexes.

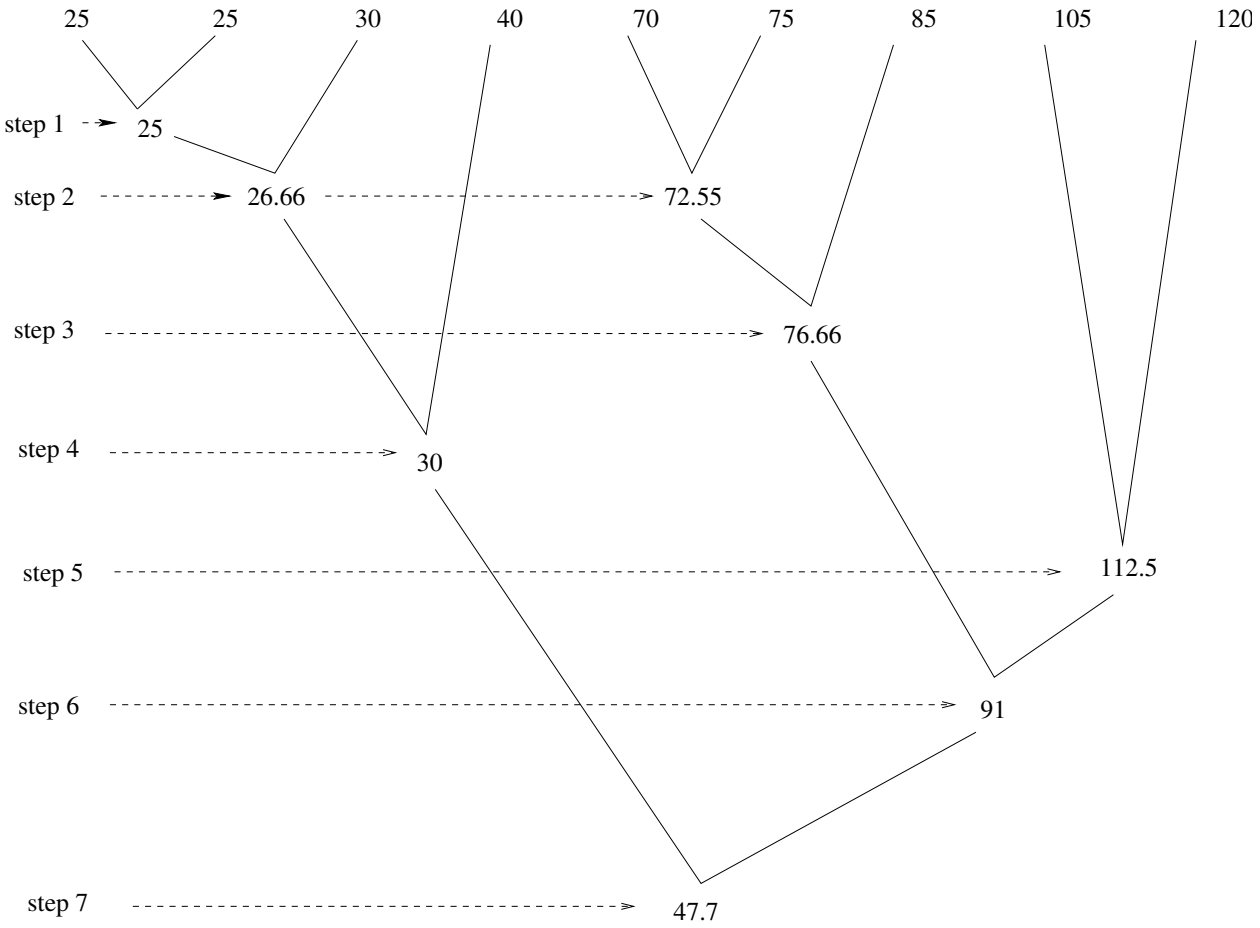

Figure 2: Dendogram for Problem 9