1. Formulate the following XPath queries for the document in Figure 17.17:

   (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
   ```

2. 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:value-of select="."/>
        </xsl:if>
     </xsl:template>
   </xsl:stylesheet>
   ```

3. 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>
```

4. Use the document structure described in exercise 17.21 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:

```
<EasyClasses>
(
  FOR $c IN
     distinct(
         filter(document("http://xyz.edu/transcript.xml")//tuple,
                .|CrsCode|Semester)
             )
  LET $grades :=
           document("http://xyz.edu/transcript.xml")
                 //tuple[CrsCode/@value=$c/CrsCode/@value
                         and Semester/@value=$c/Semester/@value]
                      /Grade/@value
  WHERE   EVERY $g IN $grades
            SATISFIES numericGrade($g) >= numericGrade("B")
  RETURN $c
)
</EasyClasses>
```

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

   Solution:

```
<GoodStudents>
(
  FOR $sid IN
     distinct(
         filter(document("http://xyz.edu/transcript.xml")//tuple,
```

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

5. 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"
DEFINE FUNCTION maxBranching($n)  RETURNS xsd:integer
  {
       LET $children := ./node()
       LET $maxChildBranching := max(maxBranching($children))
       LET $childCount := count($children)
       RETURN
            IF $childCount >= $maxChildBranching
            THEN $childCount
            ELSE $maxChildBranching
  }
```

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

6. Give an example of an instance of the TRANSCRIPT relation (Figure 4.5, page 63) and a hash function on the attribute sequence $\langle$**StudId, Grade**$\rangle$ that sends two identical tuples in $\pi_{\mathtt{StudId,Semester}}(Transcript)$ 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: $\langle$111111111, EE101, F1997, A$\rangle$ and $\langle$111111111,MAT123,F1997,B$\rangle$. Since these tuples have identical projections on the attributes **StudId, Semester**, they become one in the projected relation $\pi_{\mathtt{StudId,Semester}}(Transcript)$.

   Consider now the following hash function on **StudId, Grade**:

$$f(t) \;=\; (t.\mathtt{StudId} + (t.\mathtt{Grade} - \mathtt{'A'}))\ \mathtt{mod}\ 1000$$

   The first tuple will then be sent to the bucket 111111112 mod 1000 = 112 and the second to 111111113 mod 1000 = 113.

7. Consider the expression

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

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), theuniversity 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.

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

(a) Nested loops

(b) Block-nested loops

(c) Index-nested loops

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.

Solution:

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

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

(b) Block-nested loops: Scan $\mathbf{s}$ once per each 400-page block of $\mathbf{r}$, 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 $\mathbf{s}$ is clustered or not. For the clustered case, all tuples of $\mathbf{s}$ that match a tuple of $\mathbf{r}$ are in the same disk block and require 1 page transfer (assume that at most 5 tuples of $\mathbf{s}$ match, so they all fit in one disk block). We also need to search the index once per each tuple of $\mathbf{r}$. Suppose the later takes 1 disk access. Thus, the total is

$$2,000 + (1+1) \times 40,000 = 82,000 \text{ pages}$$

In case of an unclustered index, the matching tuples of $\mathbf{s}$ can be in different blocks. As before, assume that $\mathbf{s}$ at most than 5 matching tuples per tuple in $\mathbf{r}$. Thus, the cost would be

$$2,000 + (1+5) \times 20,000 = 122,000 \text{ pages}$$

9. 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 $\mathbf{R}$, $\mathbf{S}$, and $\mathbf{T}$ *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:

- $\mathbf{R}$ must have: B (because of the join)
- $\mathbf{S}$ must have: ACD (because of π_{ACD})
- $\mathbf{T}$ 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, $\mathbf{T}$ should not have A and C, because $\pi_{ACD}(\mathbf{S})$ clearly suggests that it is expected that $\mathbf{S}$ 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 $\mathbf{R}$ 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 $\mathbf{S}$ and not of $\mathbf{T}$.

(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: $\pi_A((\pi_E(\mathbf{T}) \bowtie_{E=D} \pi_{ACD}(\mathbf{S})) \bowtie_{C=B} \mathbf{R})$

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

AGENT(<u>Id</u>, AgentName)
HOUSE(<u>Address</u>, OwnerId, AgentId)
AMENITY(<u>Address, Feature</u>)

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.

Next we can join with AGENT, which would take 1.2 page I/Os, since AGENT has an unclustered index on `Id`. This will still result in 5 tuples, but the tuples will be about 50%

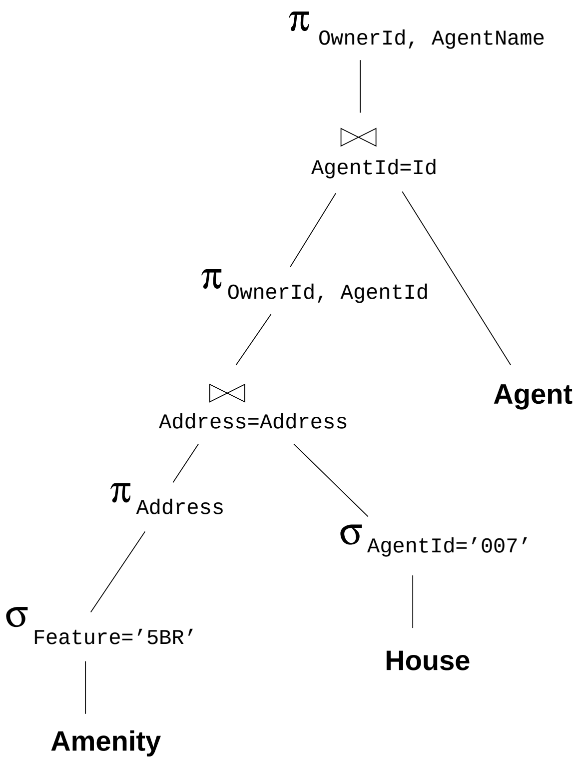
π OwnerId, AgentName
⋈ AgentId=Id
π OwnerId, AgentId
Agent
⋈ Address=Address
π Address
σ AgentId='007'
σ Feature='5BR'
House
Amenity

Figure 1:

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. 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 12 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 $6.2 + 1.2 + 12 = 19.4$ 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.

11. Show that the semijoin operation is not commutative, that is, $\mathbf{T}_1$ semijoined with $\mathbf{T}_2$ is not the same as $\mathbf{T}_2$ semijoined with $\mathbf{T}_1$.

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 $\mathbf{T}_2$, 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, the semijoin 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 $\mathbf{T}_2$

12. Design a query plan for the join example in Section 18.3.1, assuming the same table sizes as in that section, but with the following difference: An application at site B requested the join.

Solution:

1. If we send STUDENT to site C, compute the join there, and then send the result to site B, we have to send 540,000 bytes ($= 15{,}000 * 12 + 20{,}000 * 18$).

2. If we send TRANSCRIPT to site B and compute the join there, we have to send 300,000 bytes ($= 20{,}000 * 15$).

Thus alternative 2 is better. If we also allow the alternative of the semijoin described in that section, except that after the join is performed in step 3 of that protocol, the result is sent to site B, we have to send 465,000 bytes ($= 45{,}000 + 60{,}000 + 360{,}000$), as described in that section Thus alternative 2 is still the best.

13. Design SQL queries for the supermarket example that will return the information needed to make a table similar to that of Figure 19.10, except that markets are aggregated by state and time is aggregated by months.

(a) Use CUBE or ROLLUP operators.

Solution:

```
SELECT M.Region, 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.Region, T.month)
```

(b) Do not use CUBE or ROLLUP operators.

Solution:

```
SELECT  M.Region, 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.Region, T.month

SELECT  M.Region, SUM(S.Sales_Amt)
FROM    SALES S, MARKET M
WHERE   S.Market_Id = M.Market_Id
GROUP BY M.Region

SELECT  T.month, SUM(S.Sales_Amt)
FROM    SALES S, TIME T
WHERE   S.Time_Id = T.Time_Id
GROUP BY T.month
```

14. Perform the a priori algorithm on the table of Figure 19.14 to determine all reasonable two-item associations.

    Solution:

    All reasonable two-item associations are:

    $$Purchase\_diapers \rightarrow Purchase\_beer$$

    $$Purchase\_beer \rightarrow Purchase\_diapers$$