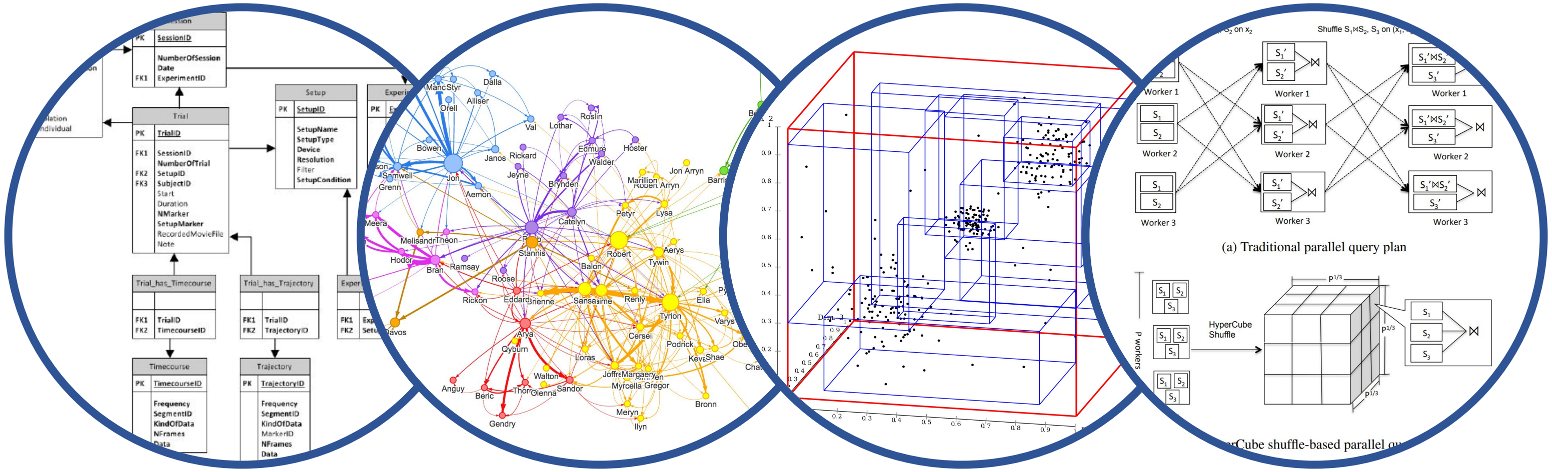


Introduction to Data Management

Database Tuning

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

HW5: Applying Transaction Logic

- HW5 midpoint was yesterday
  - If you want to take a late day, tag and then push so we know which version to use:
    git tag milestone1

HW5: Applying Transaction Logic

- Applications generally need to
  - **Check/Set isolation levels**
  - **Specify operations as transactions**
- Common mistakes/misconceptions:
  - You do not need to implement locking. The DBMS takes care of it.
  - You must **close all explicit transactions** with COMMIT or ROLLBACK. Not doing so will cause the application to hang (wait due to unfinished locking).

HW5: Applying Transaction Logic

- Applications generally need to
  - **Check/Set isolation levels**
  - **Specify operations as transactions**
- Common mistakes/misconceptions:
  - You do not need to implem
    takes care of it.
  - You must **close all exp**
    COMMIT or ROLLBACK. Not c
    application to hang (wait d
    locking).

We did this for you!

One close should *execute* per transaction, may have multiple in code:

```
if (failure_case) {
  rollback();
  return false;
}
...
commit();
return true;
```

Feedback from ETL assessment

What is helping you learn in this course?

- TAs in section
- Homework - aligns well with lecture, practice
- Lecture slides posted in advance
- In-class exercises (caveat: polling)

Feedback from ETL assessment

What changes could be made that would assist you in learning?

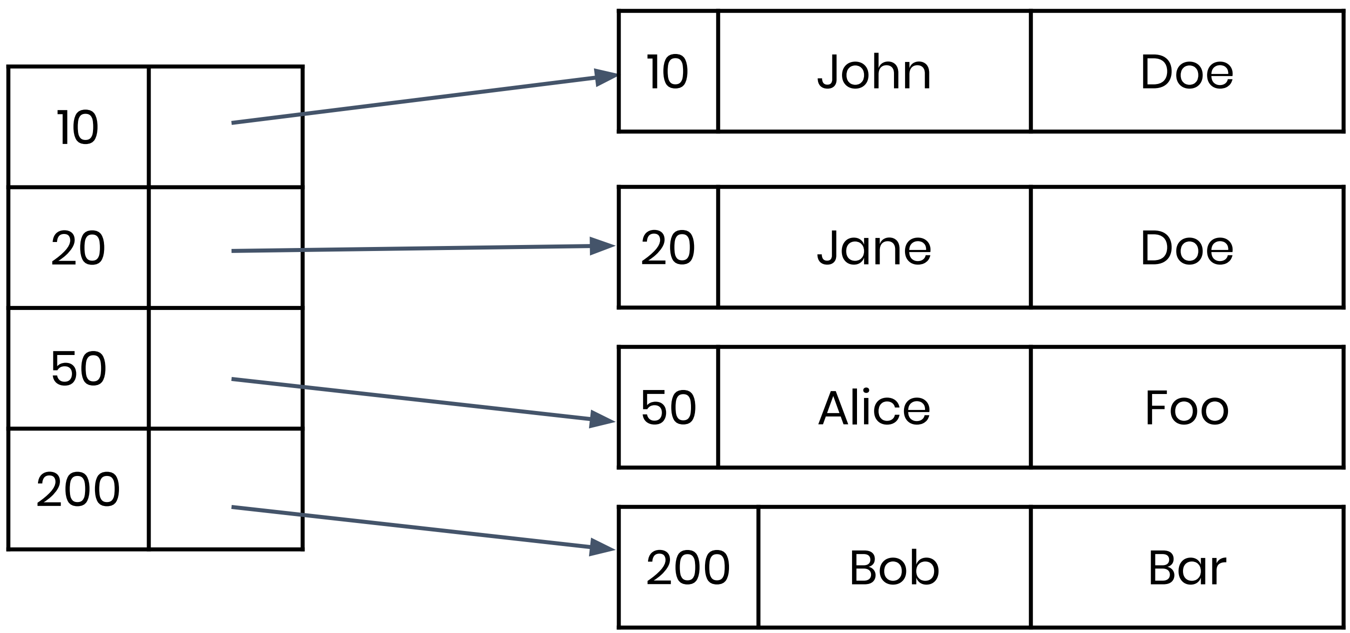
- Panopto for review
  - I'll trial it starting today - if attendance down, last-minute OH/Piazza up, I reserve right to stop
- Slide density
  - Others also want more detail in slides. I'll try some summaries, also see the recap/goal/takeaway slides
- More complex examples
  - Lengthy to fully work out in class time, we can start some examples, and see section.

Recap: Indexing

An **index** is an additional file allowing **fast access** to records given a **search key.**

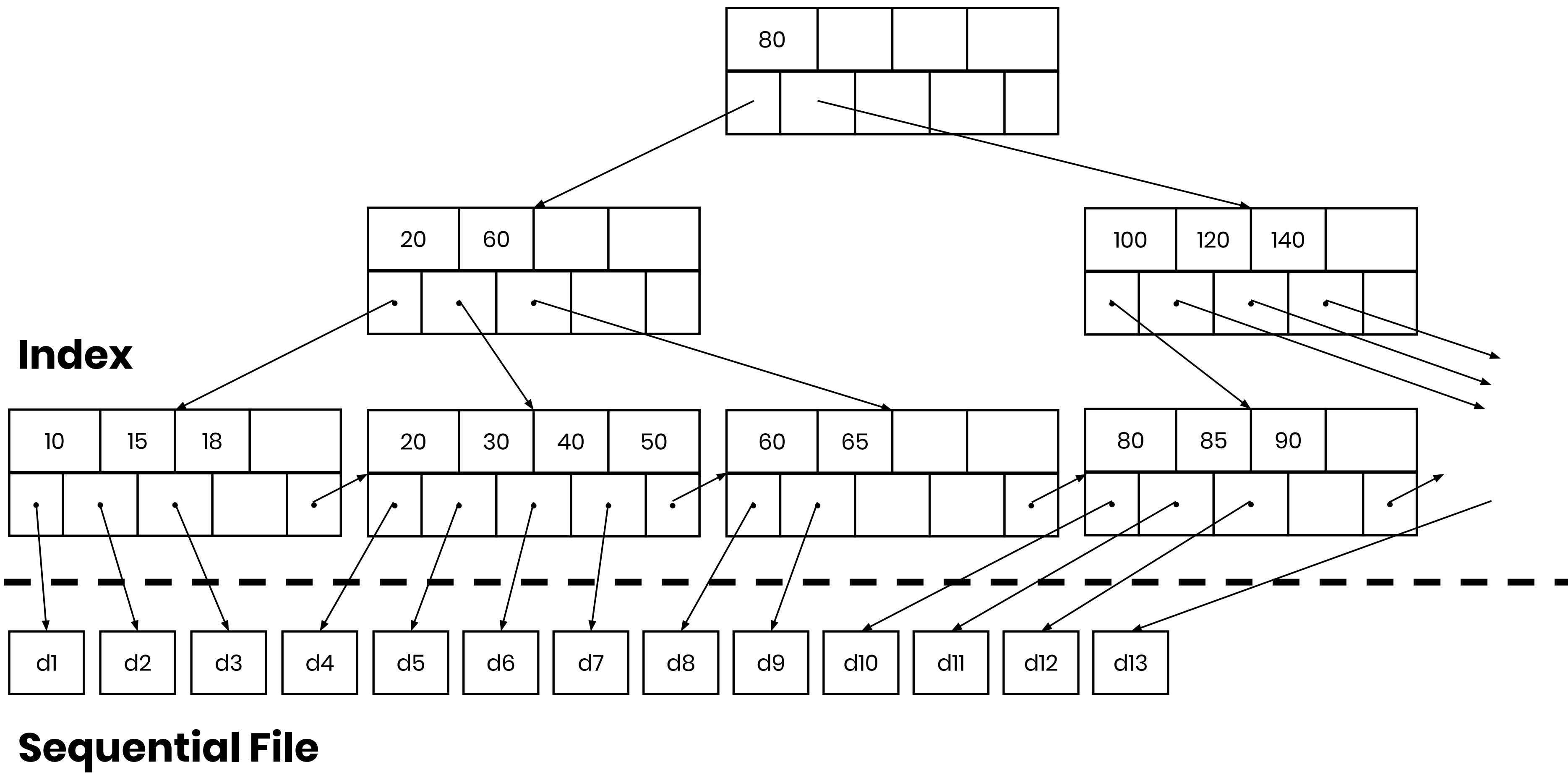
It stores **(key, value)** pairs:

(attribute, pointer to the record)



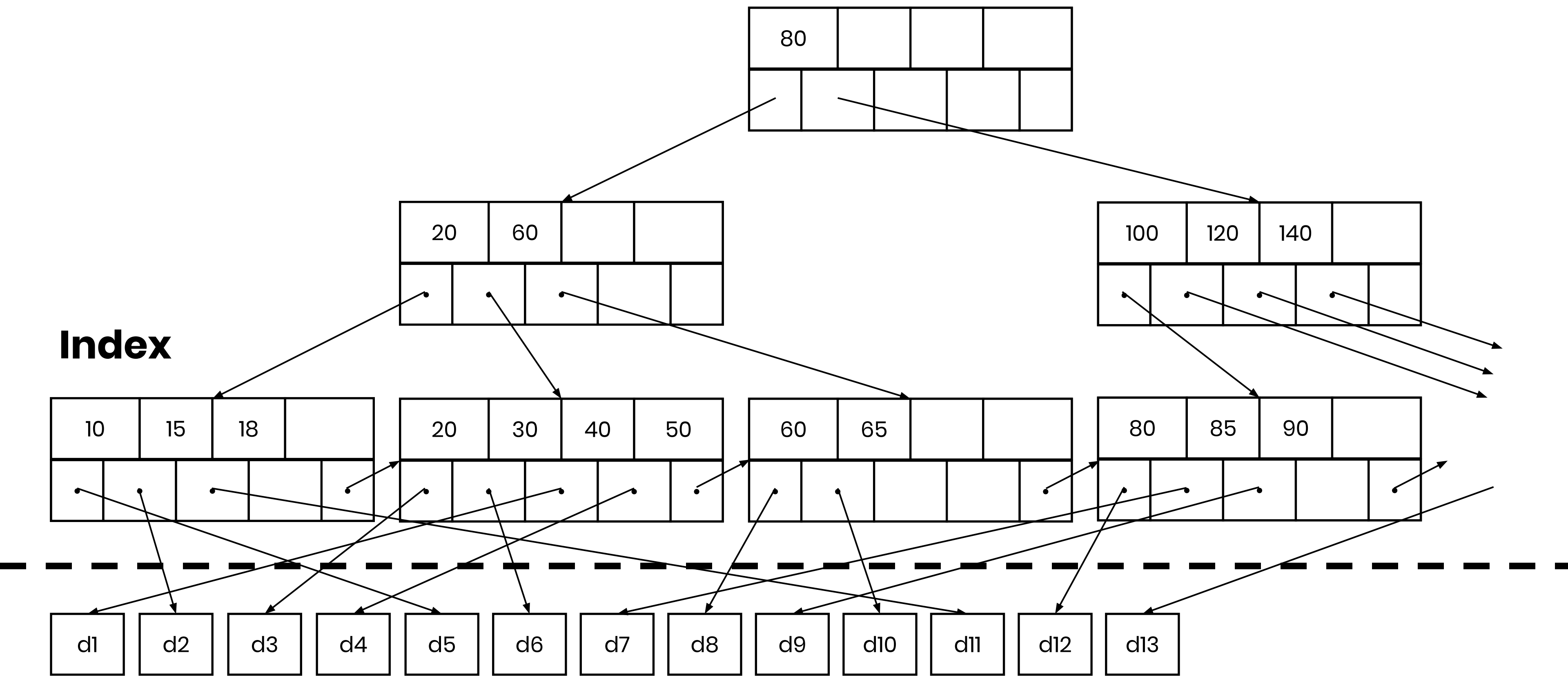
Recap: Clustered vs Unclustered Index

A **clustered index** is one that has the **same key ordering** as what is on disk (one per table)



Recap: Clustered vs Unclustered Index

- An **unclustered index** may exist without any ordering on disk (any number per table)



Sequential File with a different key or **Heap File**

Recap – Making Cost Estimations

- RDBMS keeps statistics about our tables
  - **B(R) = # of blocks** in relation R
  - **T(R) = # of tuples** in relation R
  - **V(attr, R) = # of distinct values** of attr in R

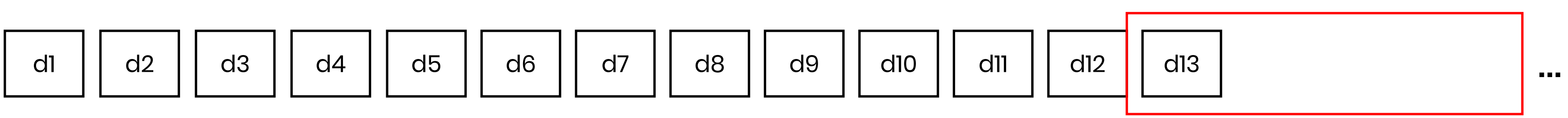
Recap: Selectivity factor

- ***Selectivity Factor*** (X) → Proportion of total data needed
- Assuming uniform distribution of data values on numeric attribute a in table R, if the condition is:
  - a=c → $X \cong \frac{1}{V(a,R)}$
  - a<c → $X \cong \frac{c-\min(a,R)}{\max(a,R)-\min(a,R)}$
  - c1<a<c2 → $X \cong \frac{c2-c1}{\max(a,R)-\min(a,R)}$
  - cond1 AND cond2 → $X \cong X_1 * X_2$
- Disclaimer: More thorough selectivity estimation will use a histogram

Recap: Sequential Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.
Without an index, finding a value must be done the "old fashioned way"

Total cost: B(R)



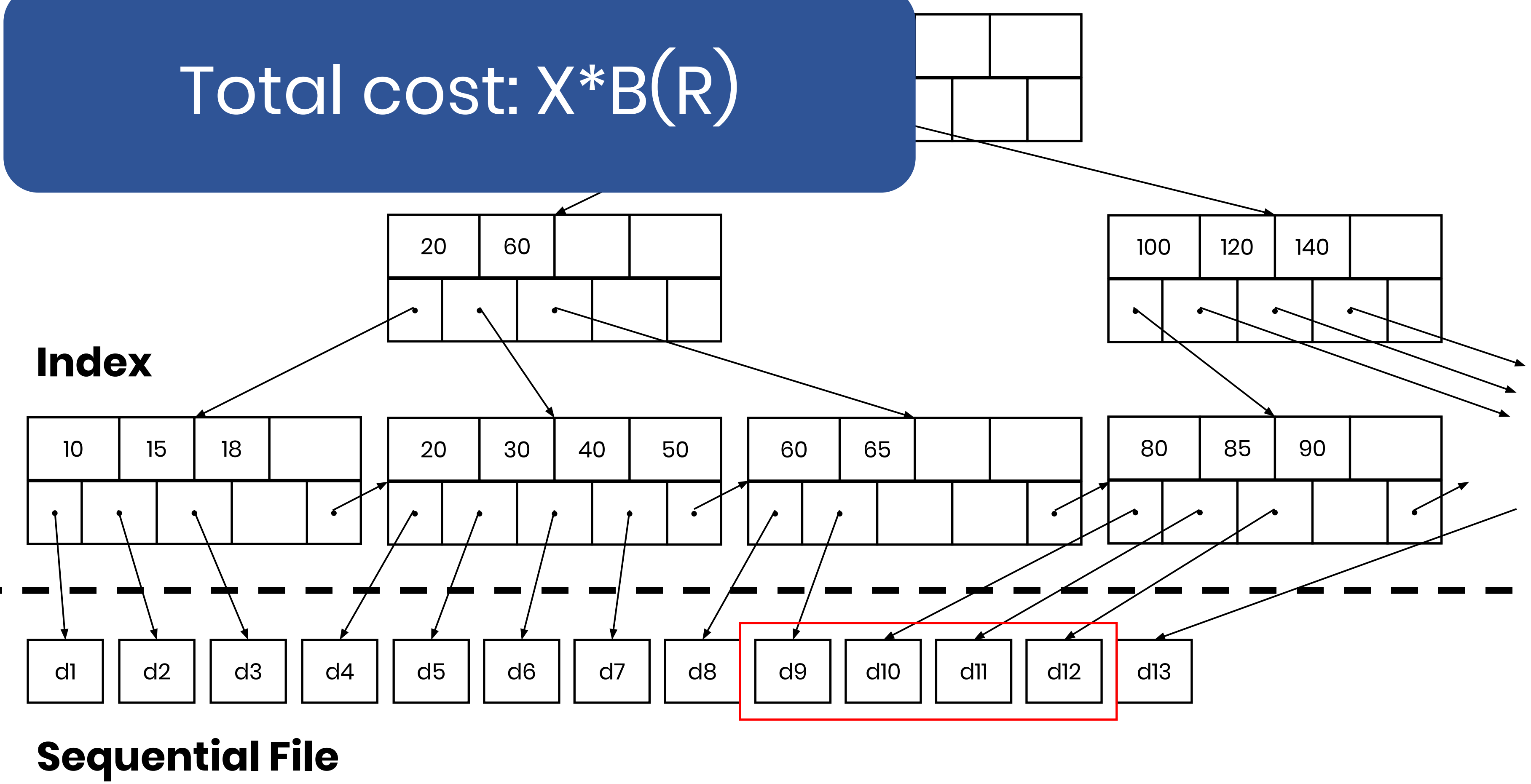
Disk

Recap: Clustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

With a clustered index, I start scanning blocks in the range they are at

Total cost: X*B(R)

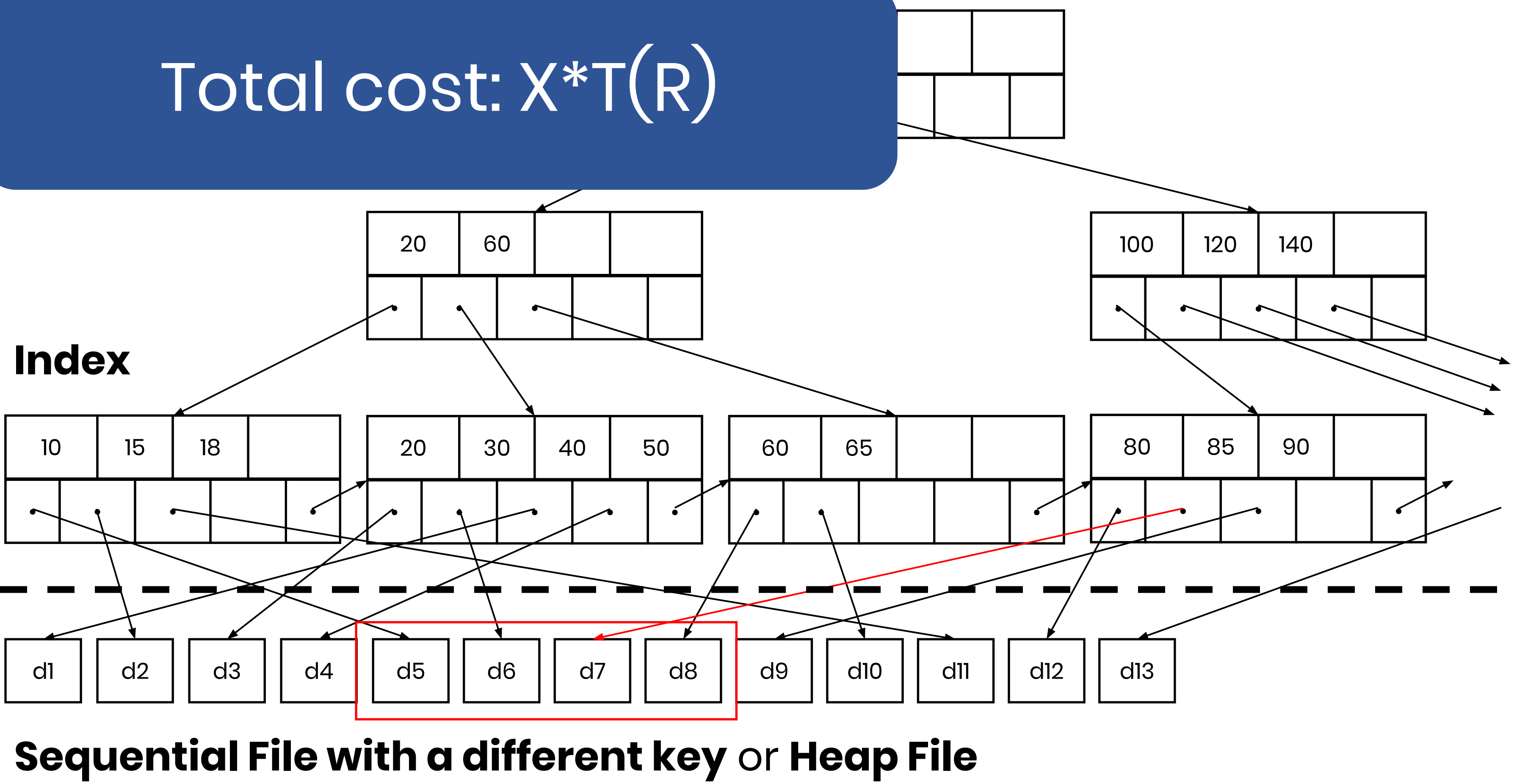


Recap: Unclustered Index Scan

Assume a block holds 4 tuples. I want tuples associated with values 40-85.

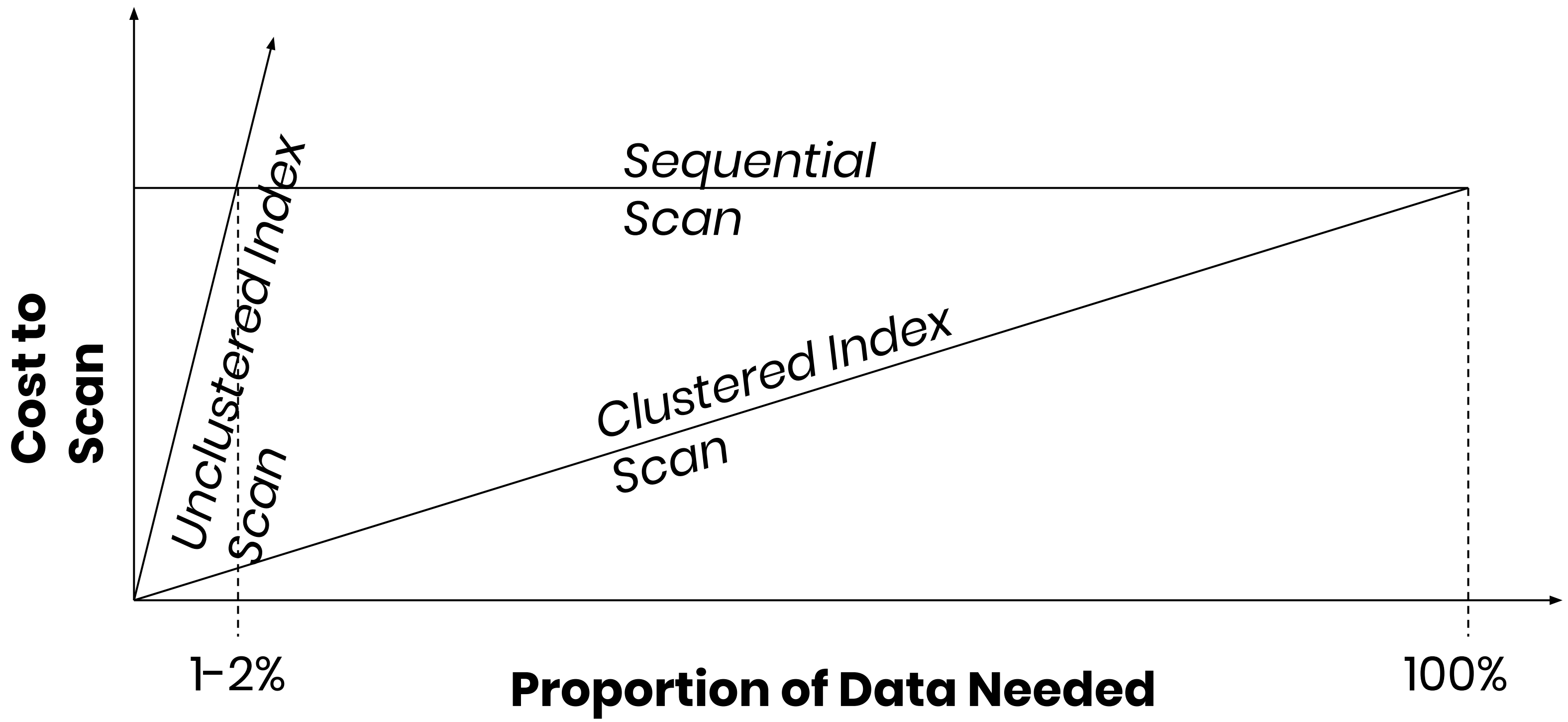
With an unclustered index, I scan tuples wherever they occur

Total cost: X*T(R)



Recap: Index Expectations

- **Sequential disk reads are faster than random ones**
  - Cost ~1–2% random scan = full sequential scan



Goals for Today

- We learned about index structures and selectivity…
- …now we’ll practice cost estimation using index-based joins.

Outline

- Database tuning
- Index join cost estimation
- Multiple joins cost estimation

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);

CREATE INDEX U_age ON Users(age)



CREATE INDEX U_age_score ON Users(age, score)



CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);

CREATE INDEX U_age ON Users(age)
```

Unclustered by default

```
CREATE INDEX U_age_score ON Users(age, score)

CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);

CREATE INDEX U_age ON Users(age)
```

Unclustered by default

```
CREATE INDEX U_age_score ON Users(age, score)
```

Order specifies precedence in sorting

```
CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);
```

```
CREATE INDEX U_age ON Users(age)
```

Unclustered by default

```
CREATE INDEX U_age_score ON Users(age, score)
```

Order specifies precedence in sorting

```
CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Does U_age_score work for these?

```
SELECT *
FROM Users
WHERE age = 21 and score > 90;

SELECT *
FROM Users
WHERE age = 21;

SELECT *
FROM Users
WHERE score > 90;
```

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);
```

```
CREATE INDEX U_age ON Users(age)
```

Unclustered by default

```
CREATE INDEX U_age_score ON Users(age, score)
```

Order specifies precedence in sorting

```
CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Does U_age_score work for these?

```
SELECT *
FROM Users
WHERE age = 21 and score > 90;
```

```
SELECT *
FROM Users
WHERE age = 21;
```

```
SELECT *
FROM Users
WHERE score > 90;
```

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);

CREATE INDEX U_age ON Users(age)
```

Unclustered by default

```
CREATE INDEX U_age_score ON Users(age, score)
```

Order specifies precedence in sorting

```
CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Reorders data on disk! (Fails if another clustered index exists)

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);
```

```
CREATE INDEX U_age ON Users(age)
```

Unclustered by default

```
CREATE INDEX U_age_score ON Users(age, score)
```

Order specifies precedence in sorting

```
CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Reorders data on disk! (Fails if another clustered index exists)

Does U_score_age work for these?

```
SELECT *
FROM Users
WHERE age = 21 and score > 90;
```

```
SELECT *
FROM Users
WHERE age = 21;
```

```
SELECT *
FROM Users
WHERE score > 90;
```

Create Indexes in SQL

```
CREATE TABLE Users (
  id INT,
  age INT,
  score INT);
```

Can also create indexes on non-numeric data (just not discussed in this class)

```
CREATE INDEX U_age ON Users(age)
```

Unclustered by default

```
CREATE INDEX U_age_score ON Users(age, score)
```

Order specifies precedence in sorting

```
CREATE CLUSTERED INDEX U_score_age ON Users(score, age)
```

Reorders data on disk! (Fails if another clustered index exists)

Leveraging Indexes

- Often for applications, workloads can be well described
  - Canvas Gradebook
    - View grades □ query for grades by student id
  - Data visualization software (e.g. Tableau)
    - 2D plot □ query on graph axis bounds
- **Create indexes to match expected query workload**

Leveraging Indexes

Make attribute K a search key if you have queries where the WHERE clause contains:

- An exact match on K
- A range predicate on K
- A join on K

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

What indexes could we make on Users?

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

What indexes could we make on Users?

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

IDs are unique so an unclustered index would do fine.

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

What indexes could we make on Users?

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

IDs are unique so an unclustered index would do fine.

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

This range query would benefit from a clustered index on score

Only one can exist!

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

This range query would benefit from a clustered index on age

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

What indexes could we make on Users?

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

IDs are unique so an unclustered index would do fine.

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

Things to consider:

- How frequently are these queries executed?
- Do either of these queries need to be returned ASAP?
- What is the expected result size for each query?
- How often is data inserted into this table?

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

What indexes could we make on Users?

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

IDs are unique so an unclustered index would do fine.

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

Without more information, default to clustering on the index that will be used more (clustered index on score)

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

What indexes could we make on Users?

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

IDs are unique so an unclustered index would do fine.

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

Hack:

- Create a covering index primarily keyed on score
- Create a covering index primarily keyed on age

Leveraging Indexes

```
CREATE TABLE Users (
  id INT PRIMARY KEY,
  age INT,
  score INT, …);
```

Expecting 1000 exec/day

```
SELECT *
  FROM Users, Assets
 WHERE Users.id = Assets.uid
```

Expecting 1000 exec/day

```
SELECT *
  FROM Users
 WHERE Users.score > 95
```

Expecting 10 exec/day

```
SELECT *
  FROM Users
 WHERE Users.age > 21
```

What indexes could we make on Users?

IDs ... ustered
ind

Essentially a sorted copy of the table. Fast but space inefficient and table updates are slow.

Hack:

- Create a covering index primarily keyed on score
- Create a covering index primarily keyed on age

“I’m not about that code monkey life”

- Choosing how to configure a database system is an interesting (i.e. hard) problem
- A database that is used by many people will often need one or more dedicated personnel to manage it (Database Administrator)
  - Logical design (multi-team coordination)
  - Physical design (hardware and system considerations)
  - Permission management (visibility and security)
  - Integration (company acquisitions and mergers)
  - …

Onto more cost estimation….

Index-Based Equijoin

- Assume index exists on the join attribute a of S

```
SELECT *
  FROM R, S
 WHERE R.a = S.a
```

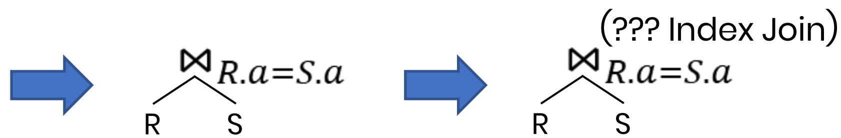
$\bowtie_{R.a=S.a}$ (R, S) → (??? Index Join) $\bowtie_{R.a=S.a}$ (R, S)

- **Clustered Index Join**
  - Perform a clustered index scan for each tuple of R
  - B(R)+T(R)(X*B(S)) = **B(R)+T(R)(<u>B(S)</u>/V(a,S))**
- **Unclustered Index Join**
  - Perform an unclustered index scan for each tuple of R
  - B(R)+T(R)(X*T(S)) = **B(R)+T(R)(<u>T(S)</u>/V(a,S))**

Index-Based Equijoin

- Assume index exists on the join attribute a of S

```
SELECT *
  FROM R, S
 WHERE R.a = S.a
```



Can't scan per block of R since tuples in blocks don't have the same attribute values

- **Clustered Index Join**
  - Perform a clustered index scan for each tuple of R
  - B(R)+T(R)(X*B(S)) = **B(R)+T(R)(<u>B(S)</u>/V(a,S))**
- **Unclustered Index Join**
  - Perform an unclustered index scan for each tuple of R
  - B(R)+T(R)(X*T(S)) = **B(R)+T(R)(<u>T(S)</u>/V(a,S))**

Multiple Joins

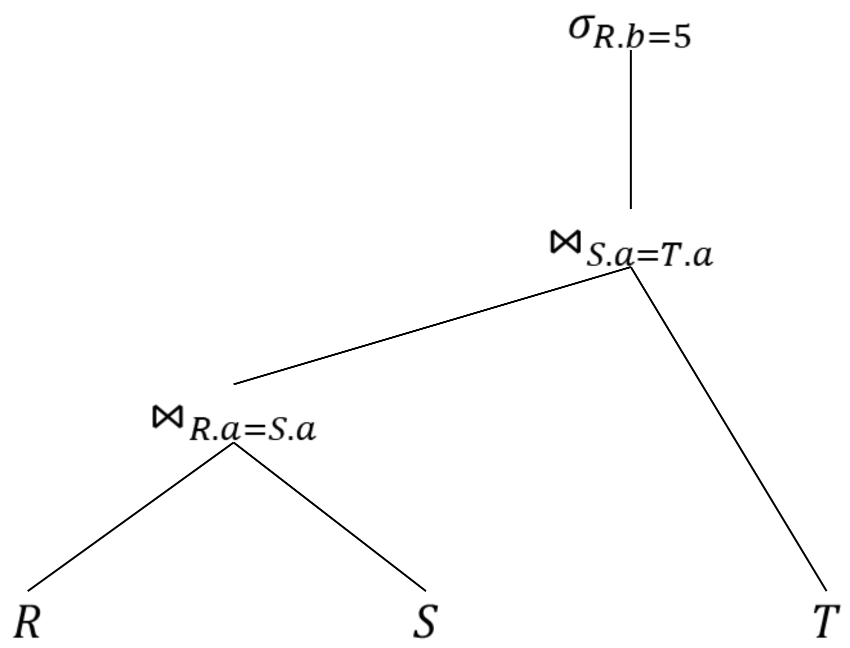
- **Pipelined Execution**
  - Tuples are processed through the entire query plan
  - Fast
- Blocking Execution
  - Subplans are computed and stored before parent operation can start
  - Simple

Pipelined Execution

- Iterator interface of RA operators (**Volcano Iterator Model**)
  - open() on every operator at start
  - next() to get the next tuple from a child operator or input table
  - close() on every operator at end

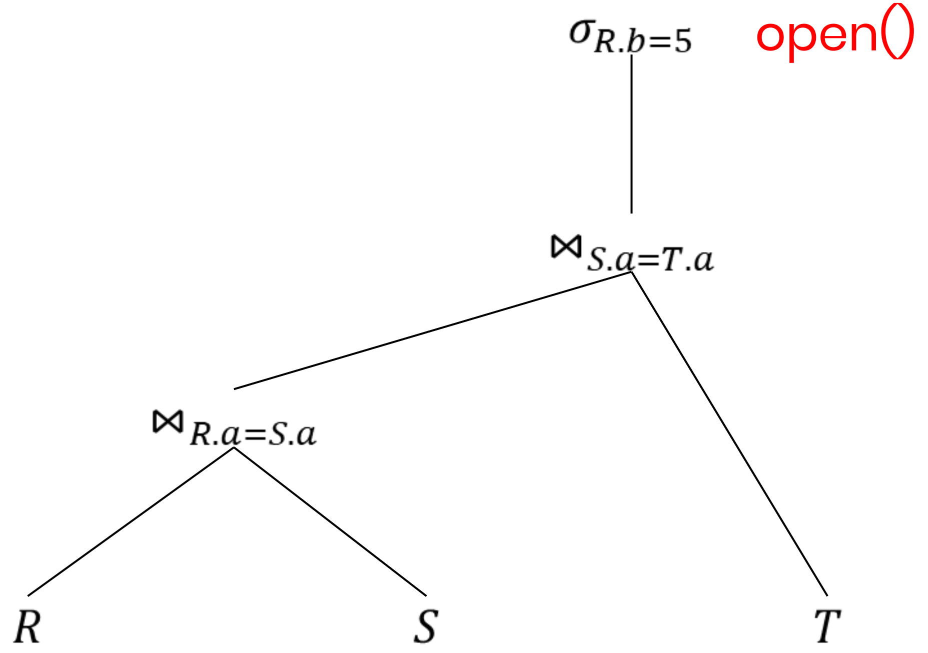
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



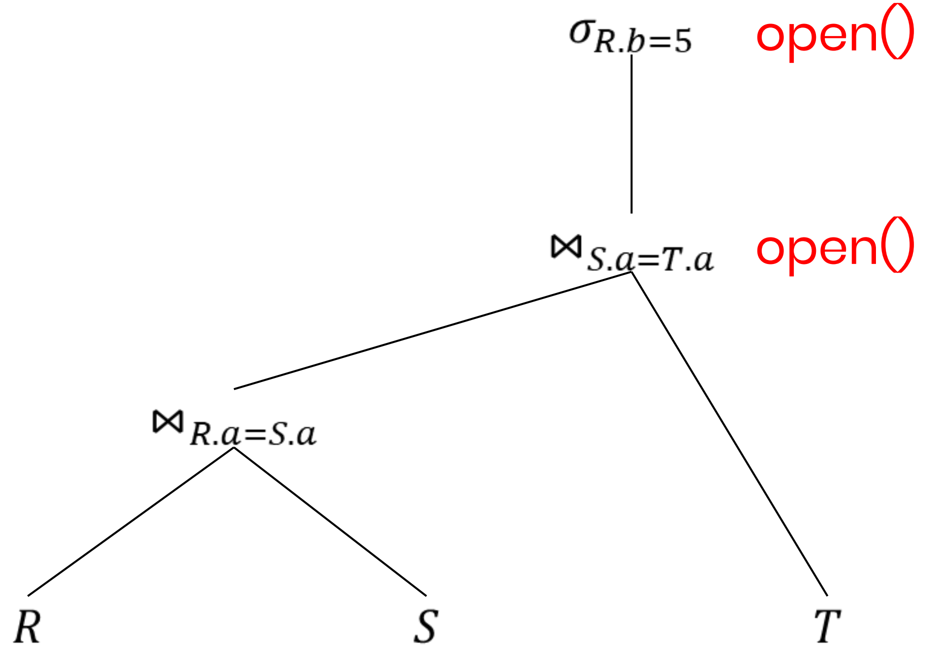
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



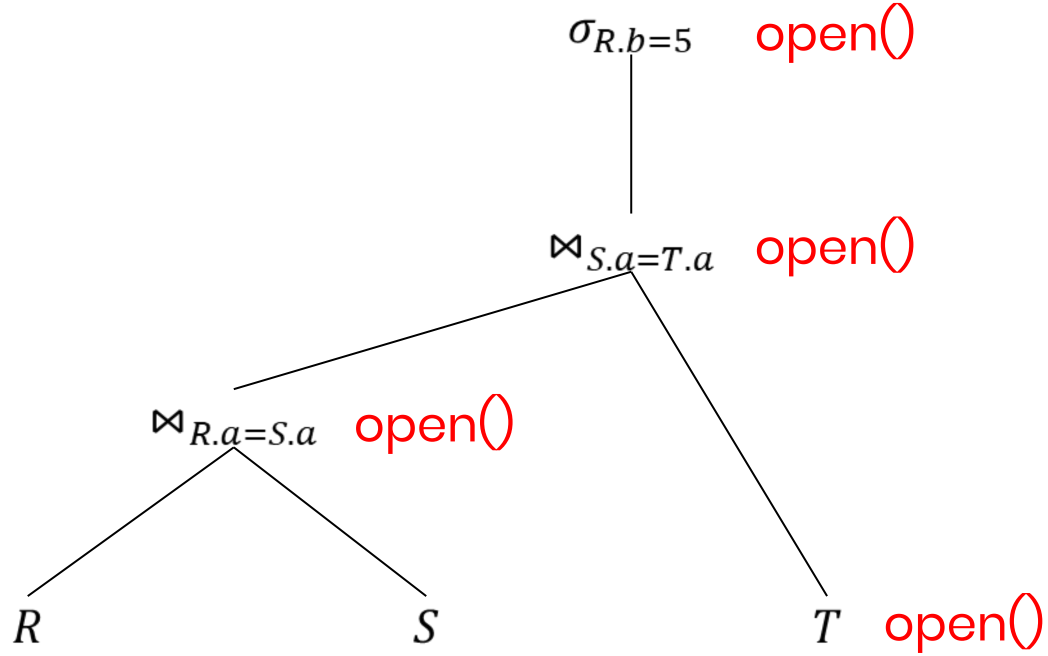
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



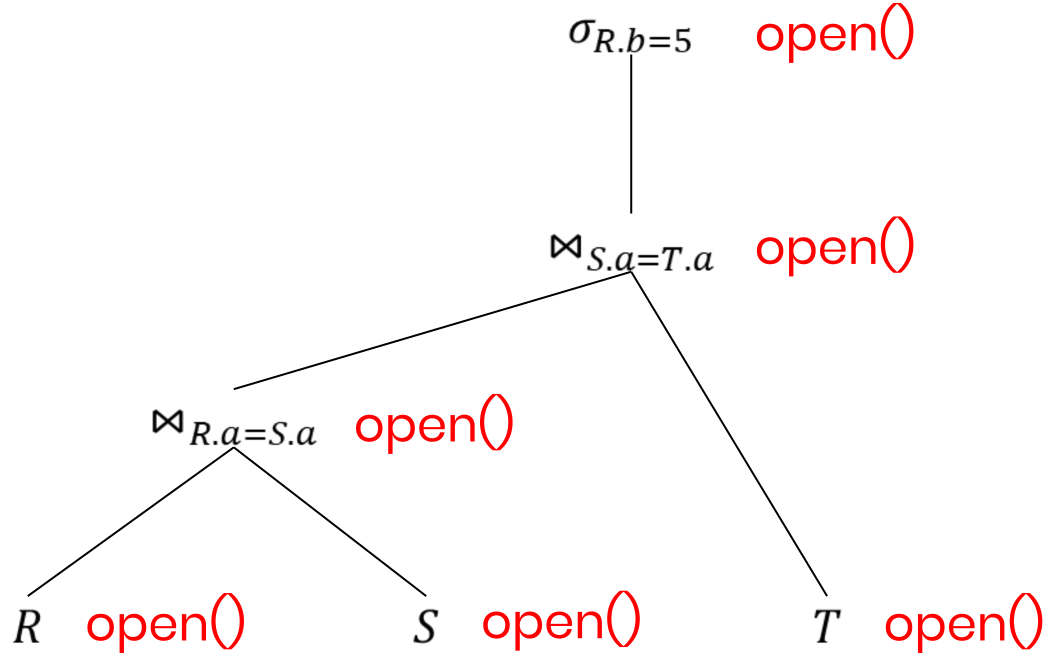
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



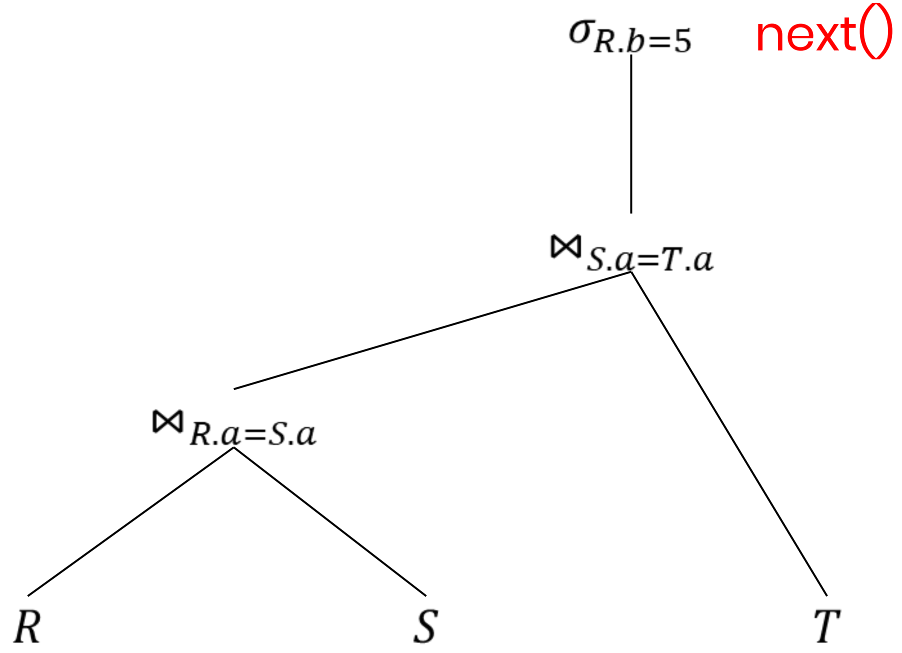
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



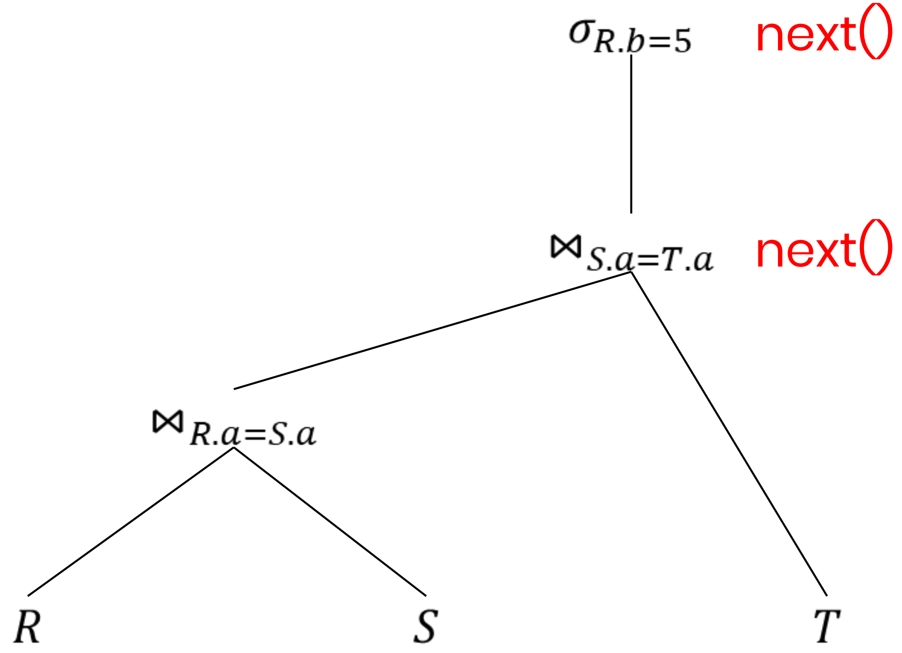
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



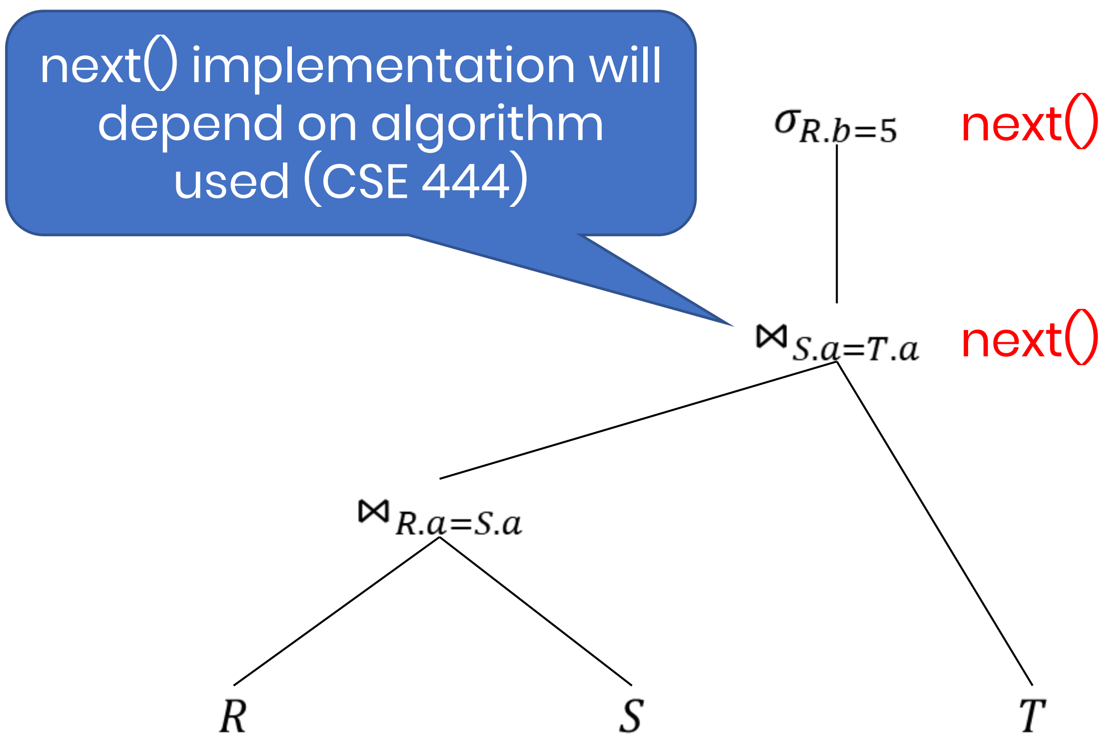
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



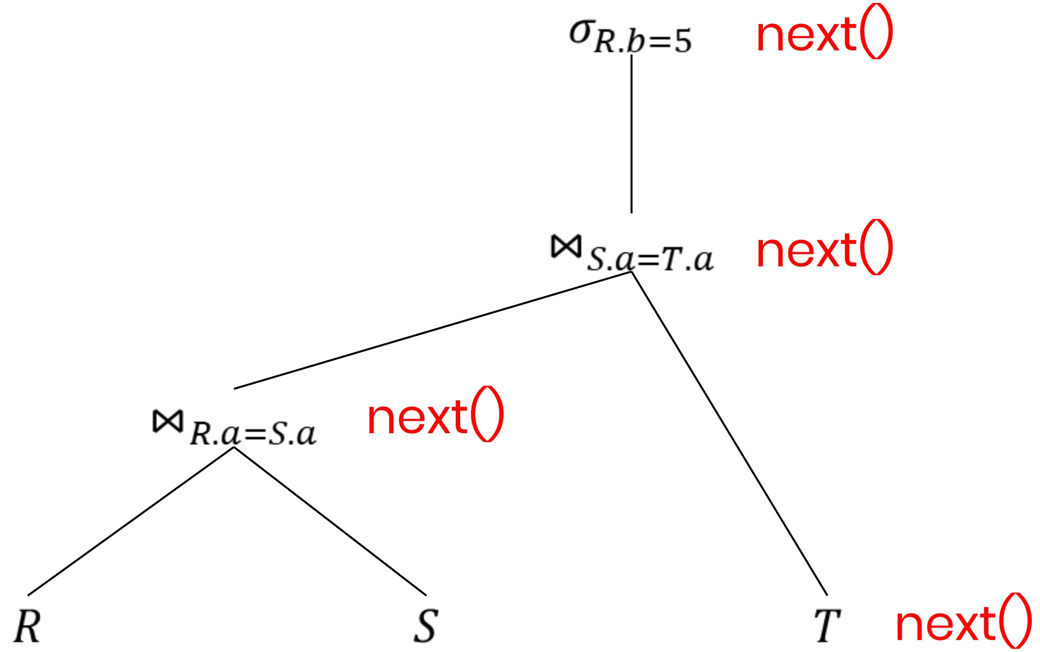
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



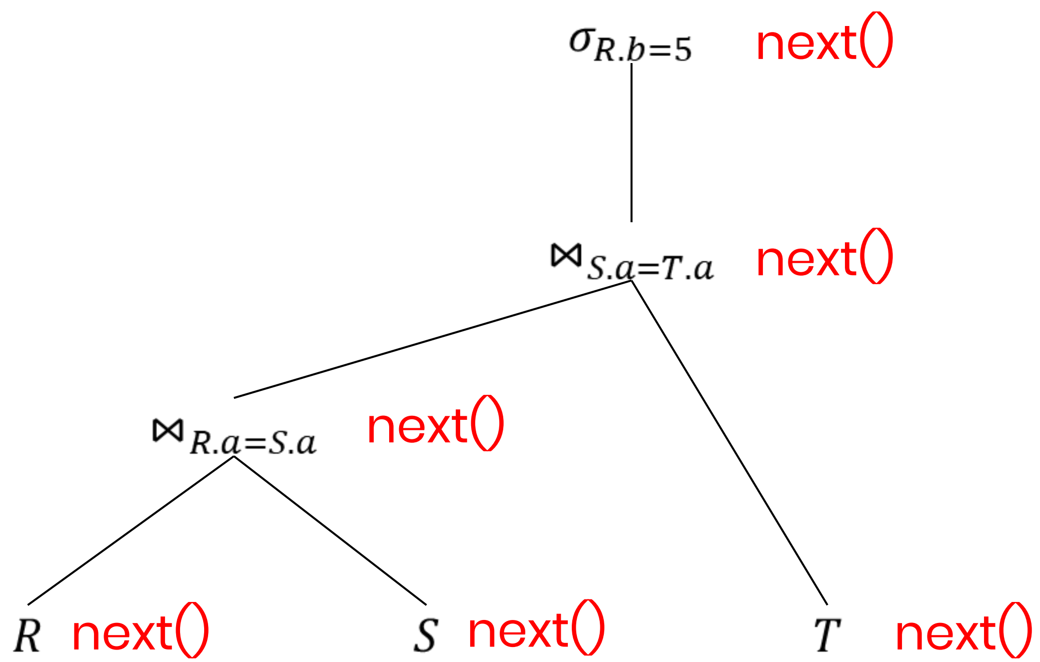
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



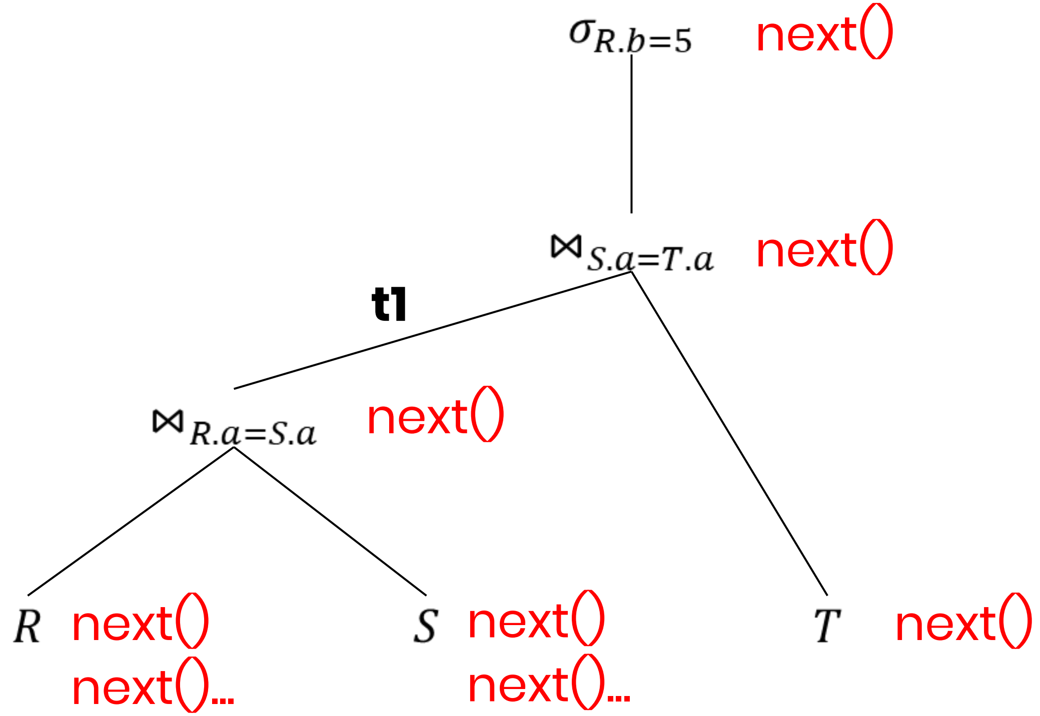
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



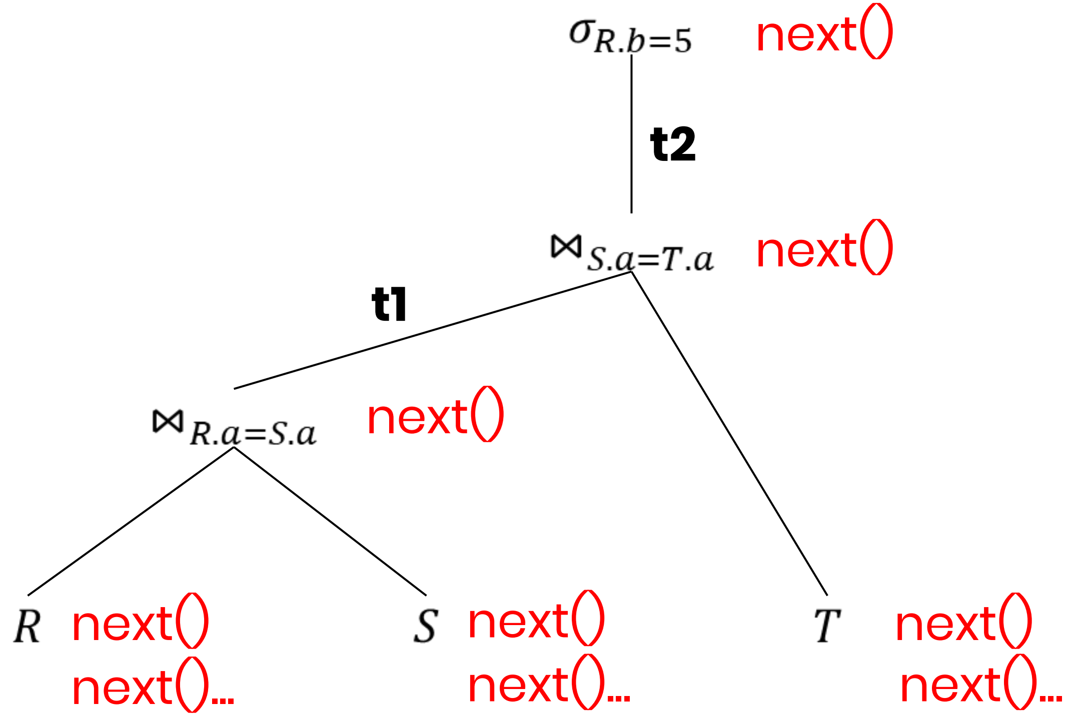
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



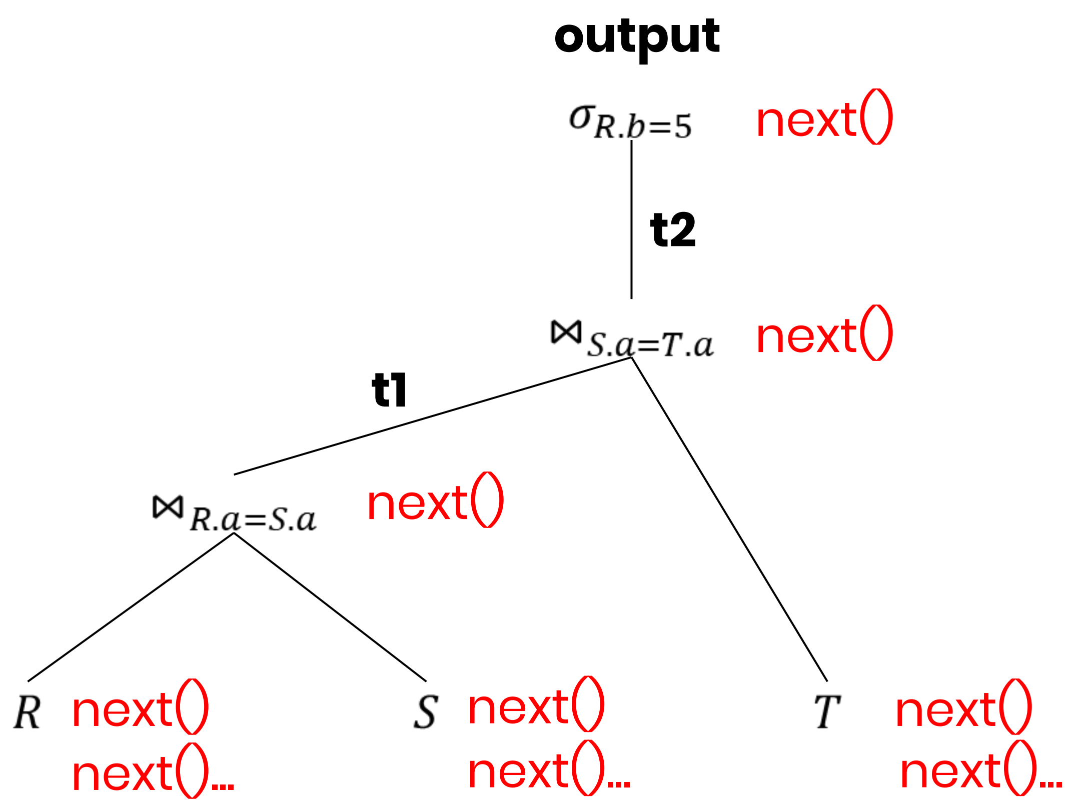
Pipelined Execution Example

- Iterator interface of RA operators (**Volcano Iterator Model**)



Pipelined Execution Example

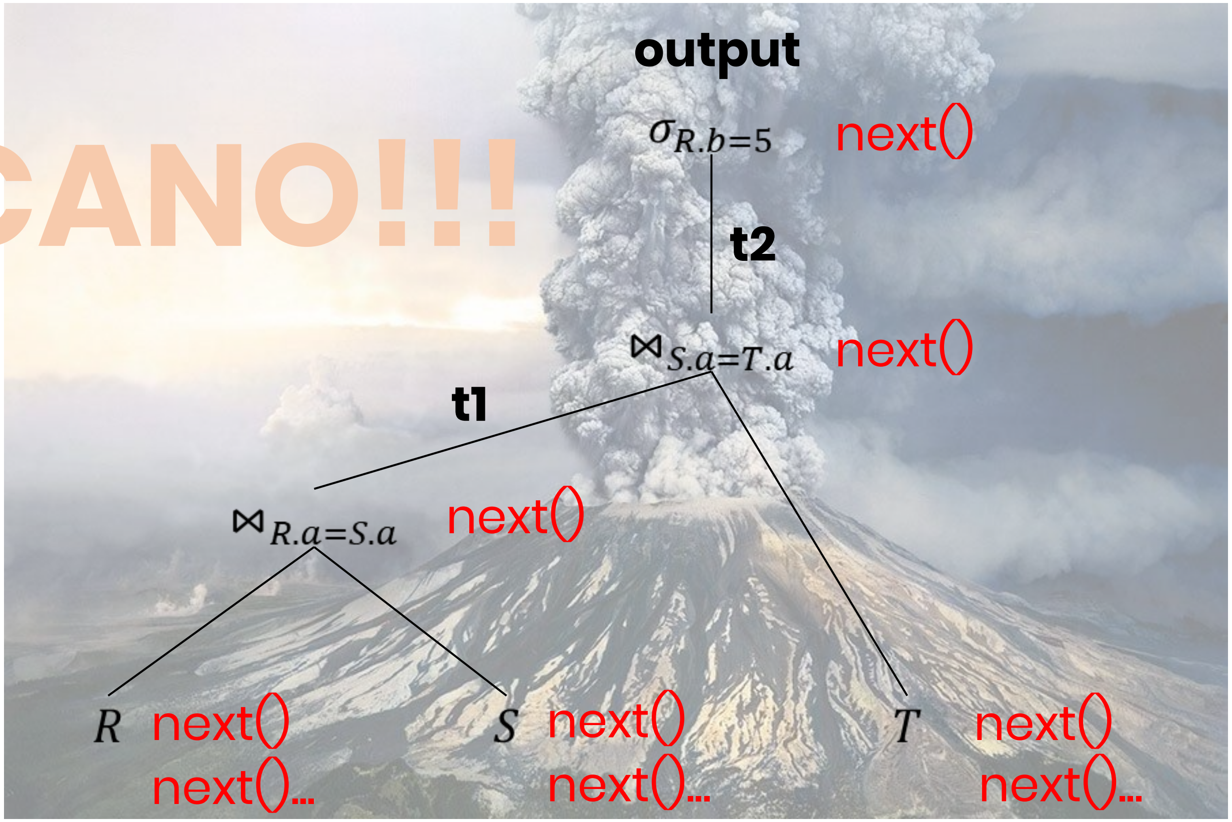
- Iterator interface of RA operators (**Volcano Iterator Model**)



Pipelined Execution Example

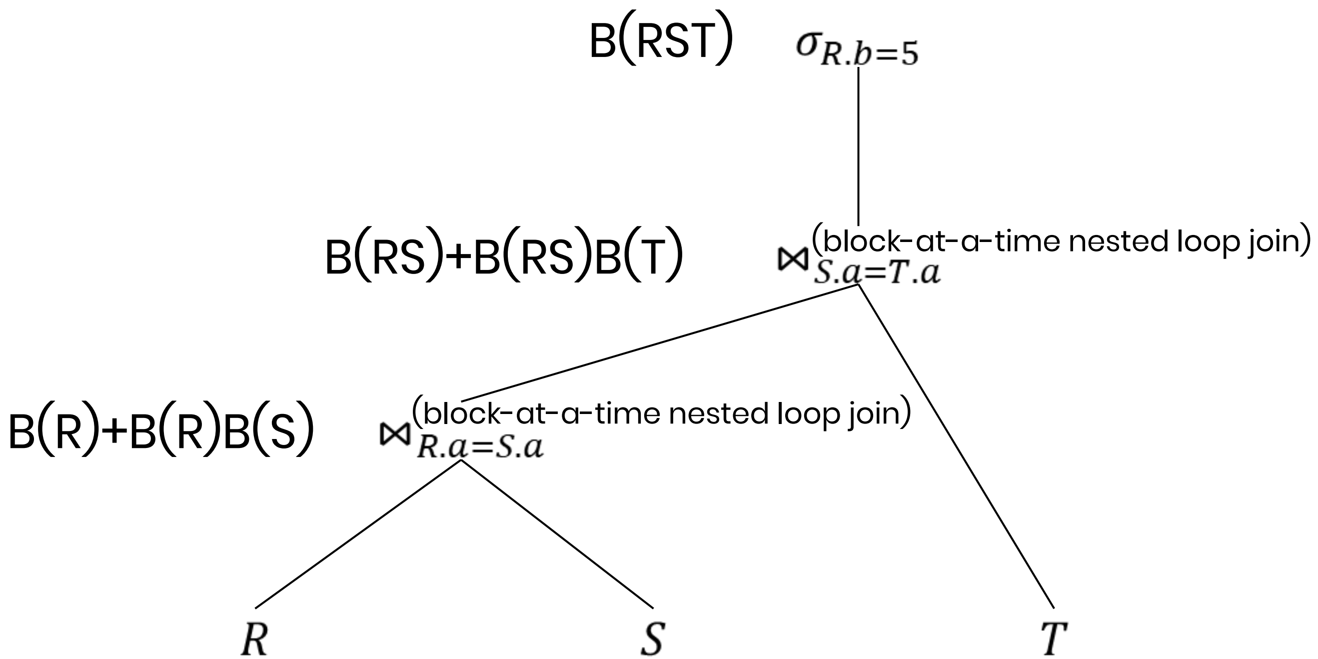
- Iterator interface of RA operators (**Volcano Iterator Model**)

VOLCANO!!!



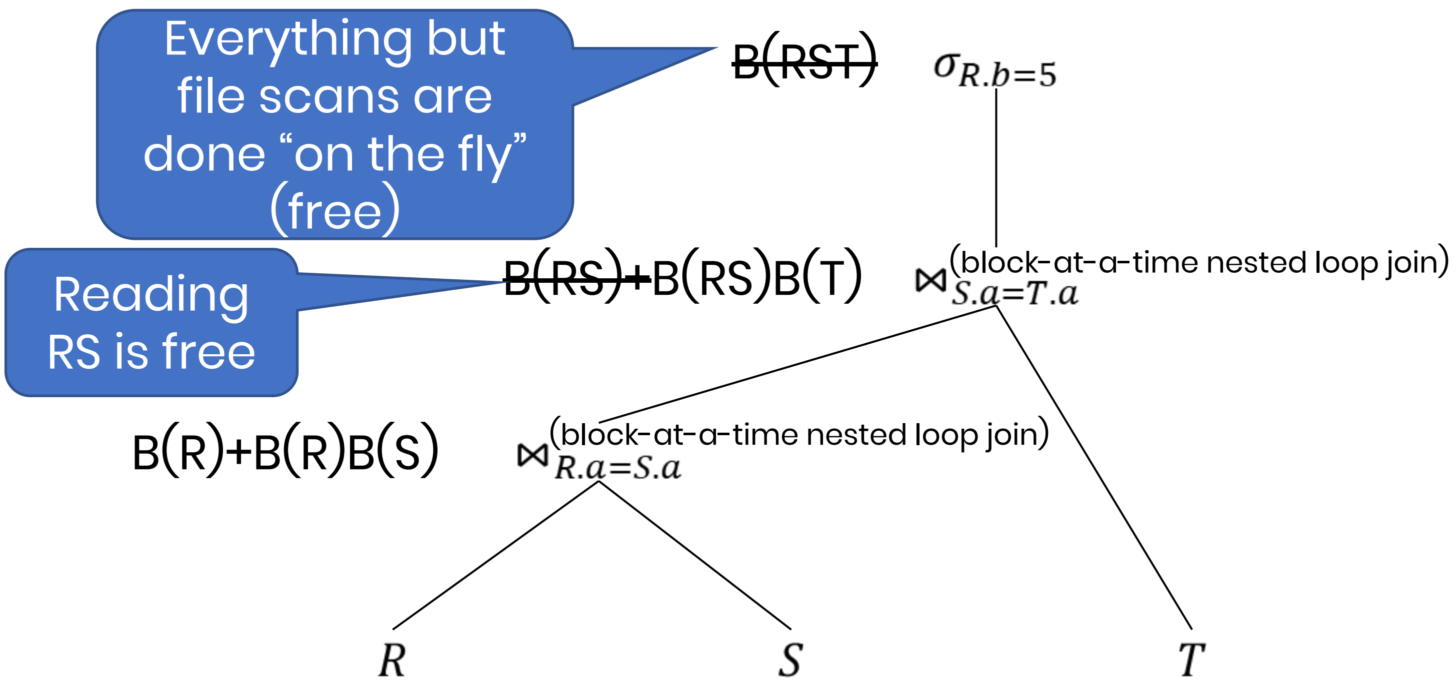
Full Plan Cost Estimation

- Estimated IO cost for a plan can be lowered under pipelined execution
- Generated tuples are in main memory so future access is "free"



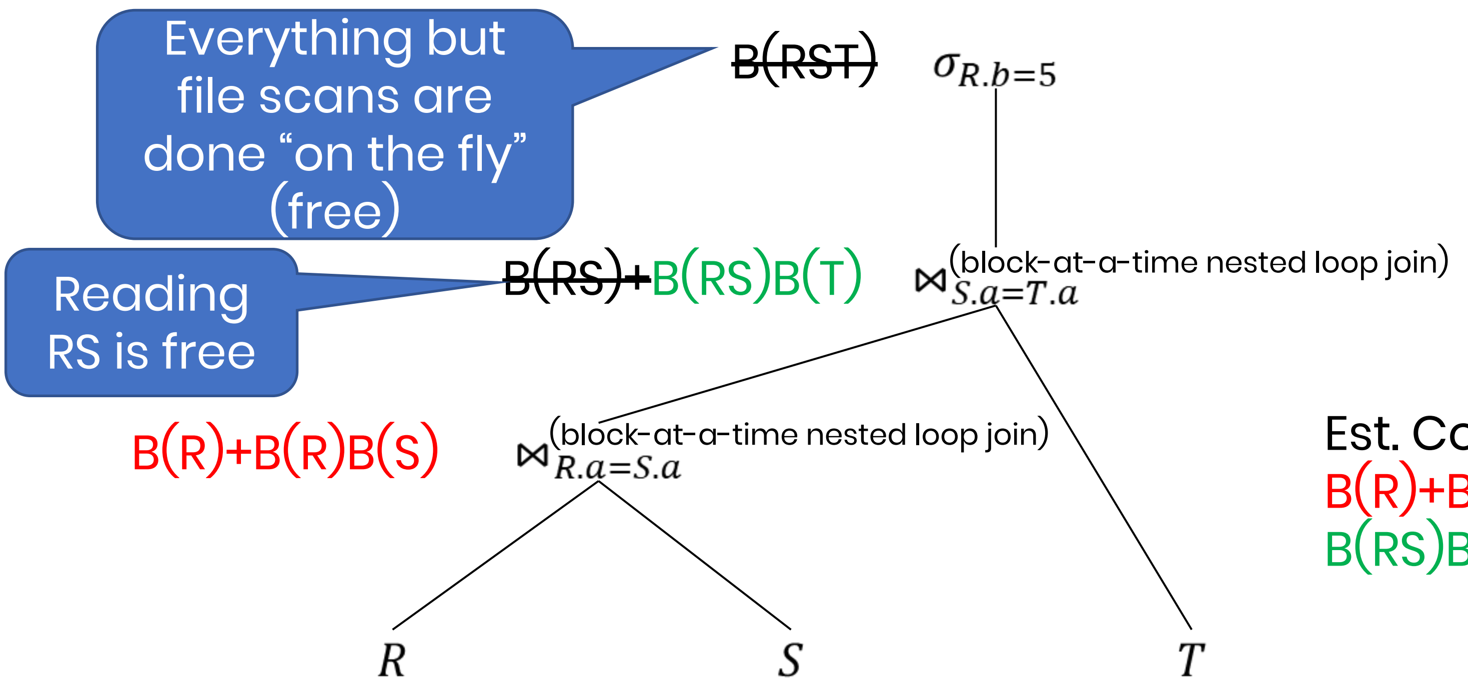
Full Plan Cost Estimation

- Estimated IO cost for a plan can be lowered under pipelined execution
- Generated tuples are in main memory so future access is “free”



Full Plan Cost Estimation

- Estimated IO cost for a plan can be lowered under pipelined execution
- Generated tuples are in main memory so future access is "free"



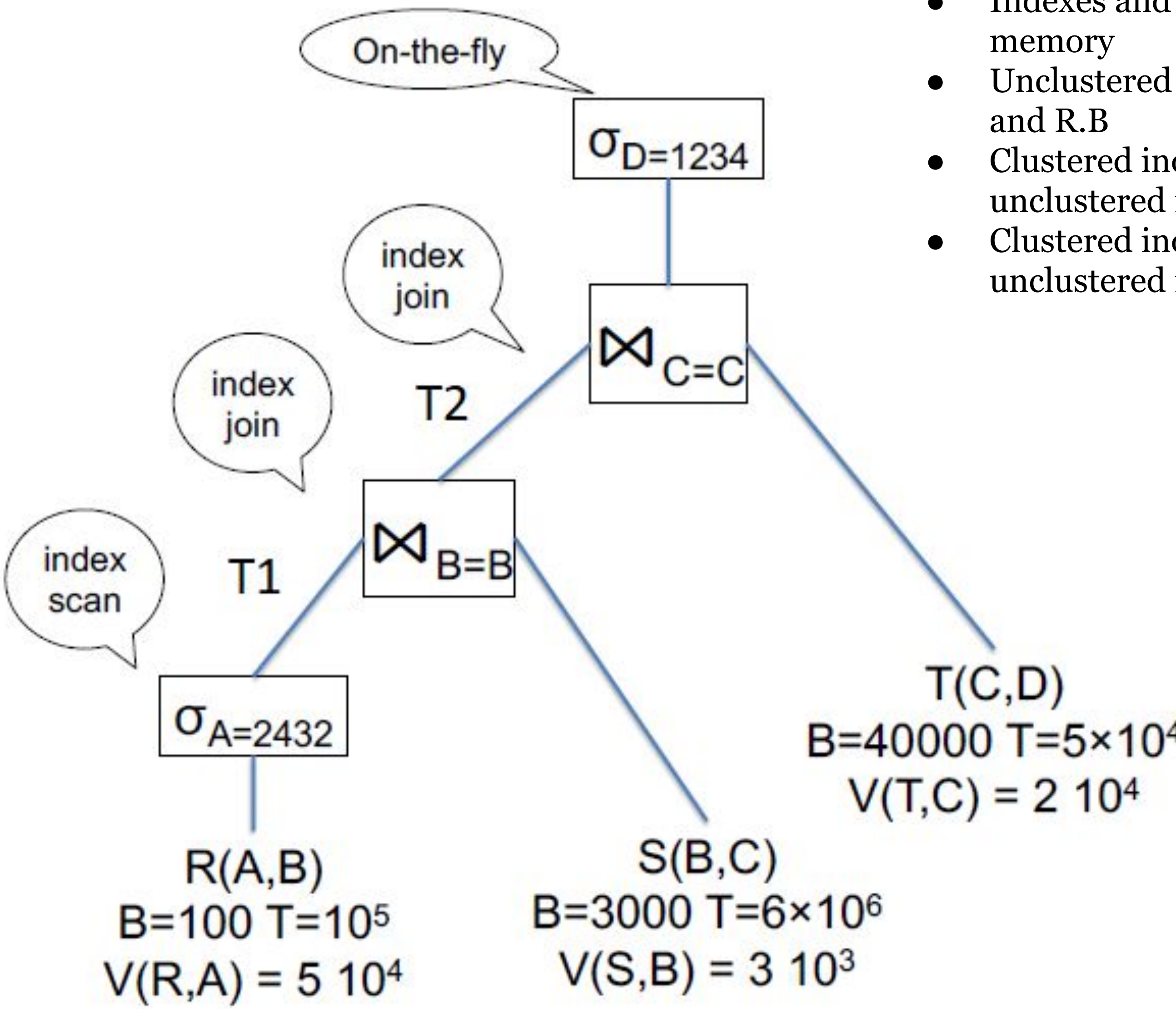
Est. Cost =
B(R)+B(R)B(S) +
B(RS)B(T)

Components of a Physical Plan

Summarizing the components we've seen:

- Access path selection for each relation
  - Scan the relation or use an index
- Implementation choice for each operator
  - Nested loop join, hash join, etc.
- Scheduling decisions for operators
  - Pipelined execution or intermediate materialization (blocking)

Example Problem



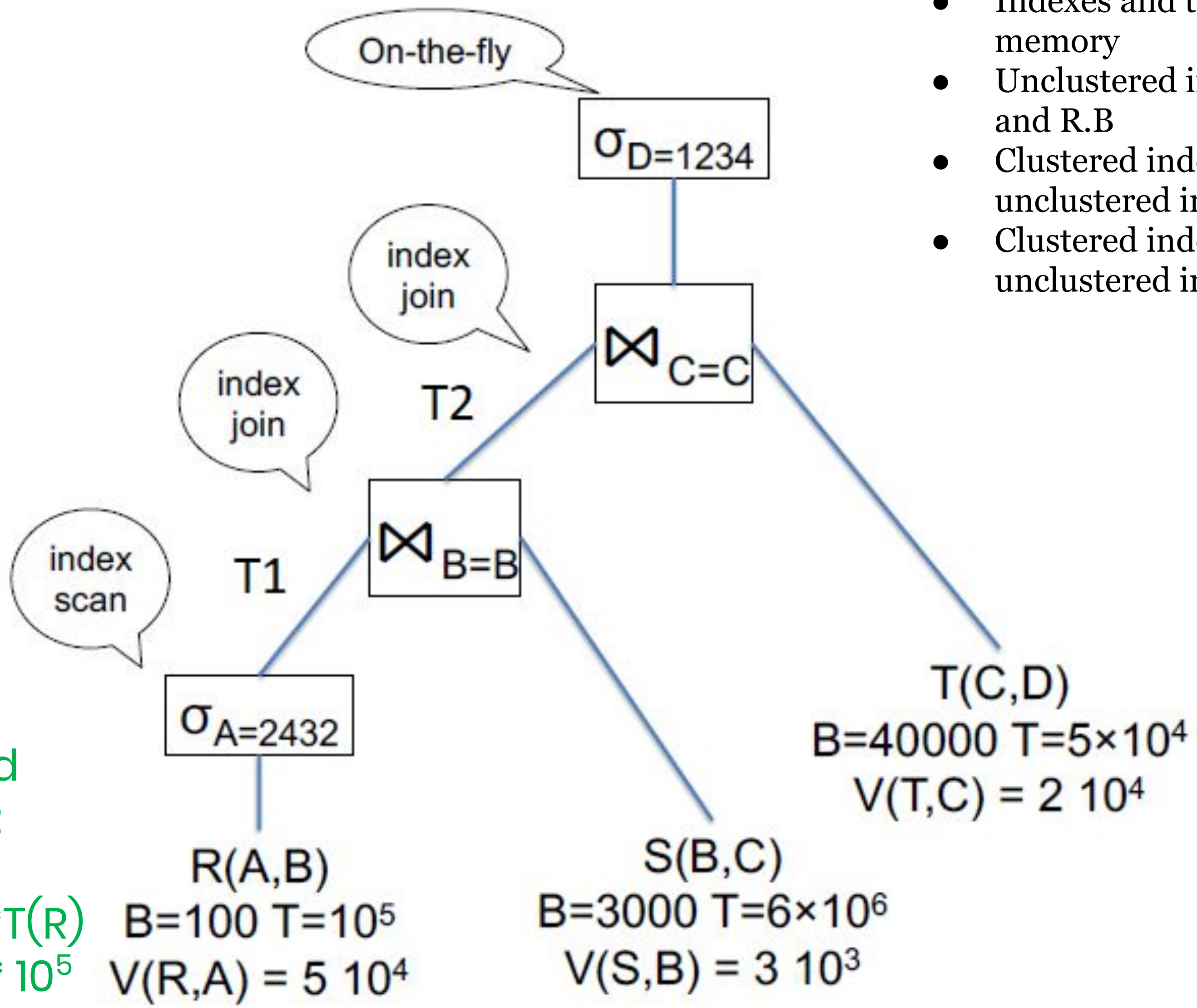
Assuming:

- Indexes and tuples fit in memory
- Unclustered indexes on R.A and R.B
- Clustered index on S.B, unclustered index on S.C.
- Clustered index on T.C, unclustered index on T.D.

Example Problem

Assuming:

- Indexes and tuples fit in memory
- Unclustered indexes on R.A and R.B
- Clustered index on S.B, unclustered index on S.C.
- Clustered index on T.C, unclustered index on T.D.



Cost of unclustered index scan:

= X*T(R)

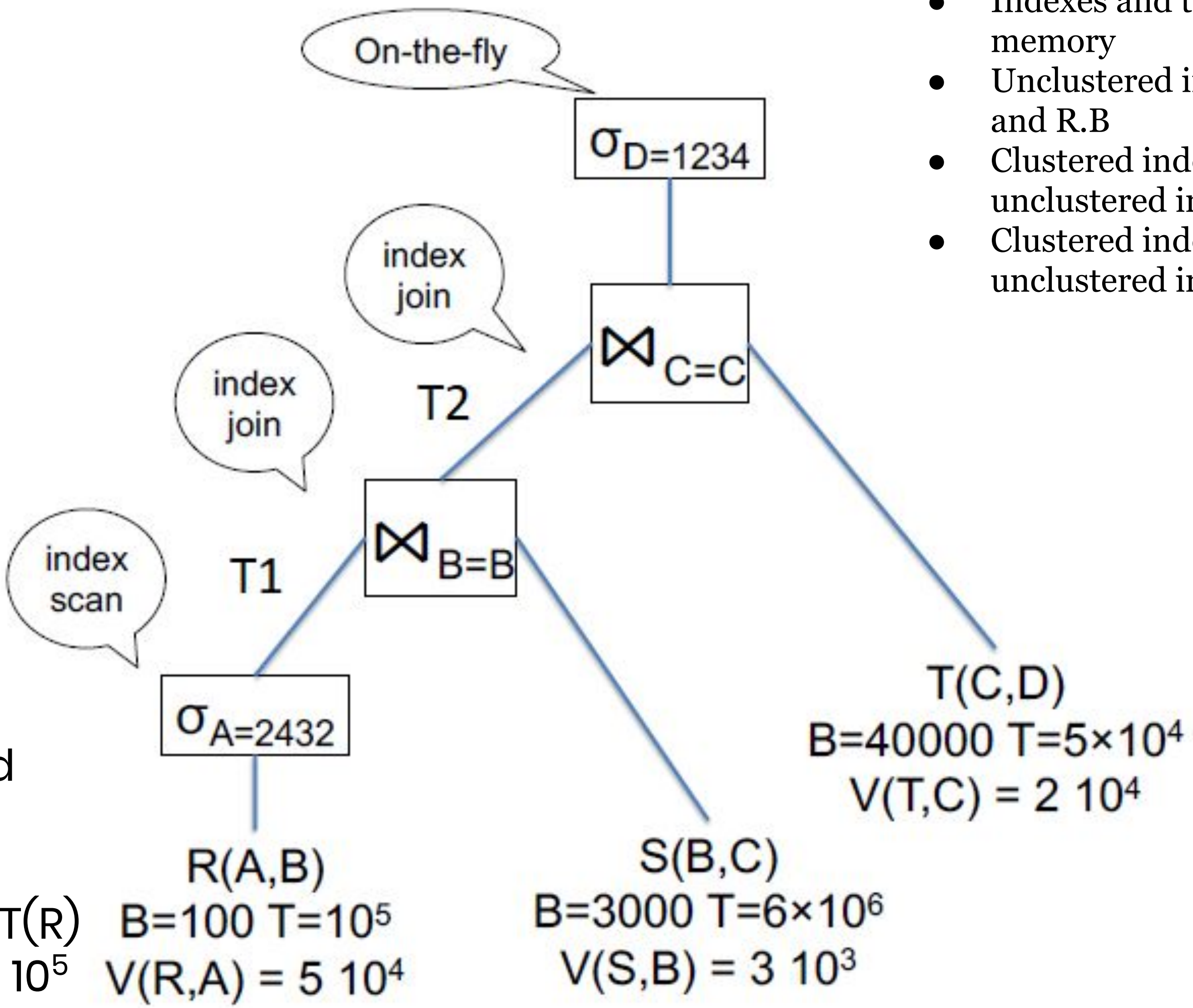
= (1/V(R,A))*T(R)

= (1/ 5*10^4) * 10^5

= 2

Example Problem

Assuming:
- Indexes and tuples fit in memory
- Unclustered indexes on R.A and R.B
- Clustered index on S.B, unclustered index on S.C.
- Clustered index on T.C, unclustered index on T.D.



T(T1) = 2

Cost of unclustered index scan:

$= X * T(R)$

$= (1/V(R,A)) * T(R)$

$= (1/ 5*10^4) * 10^5$

$= 2$

Example Problem

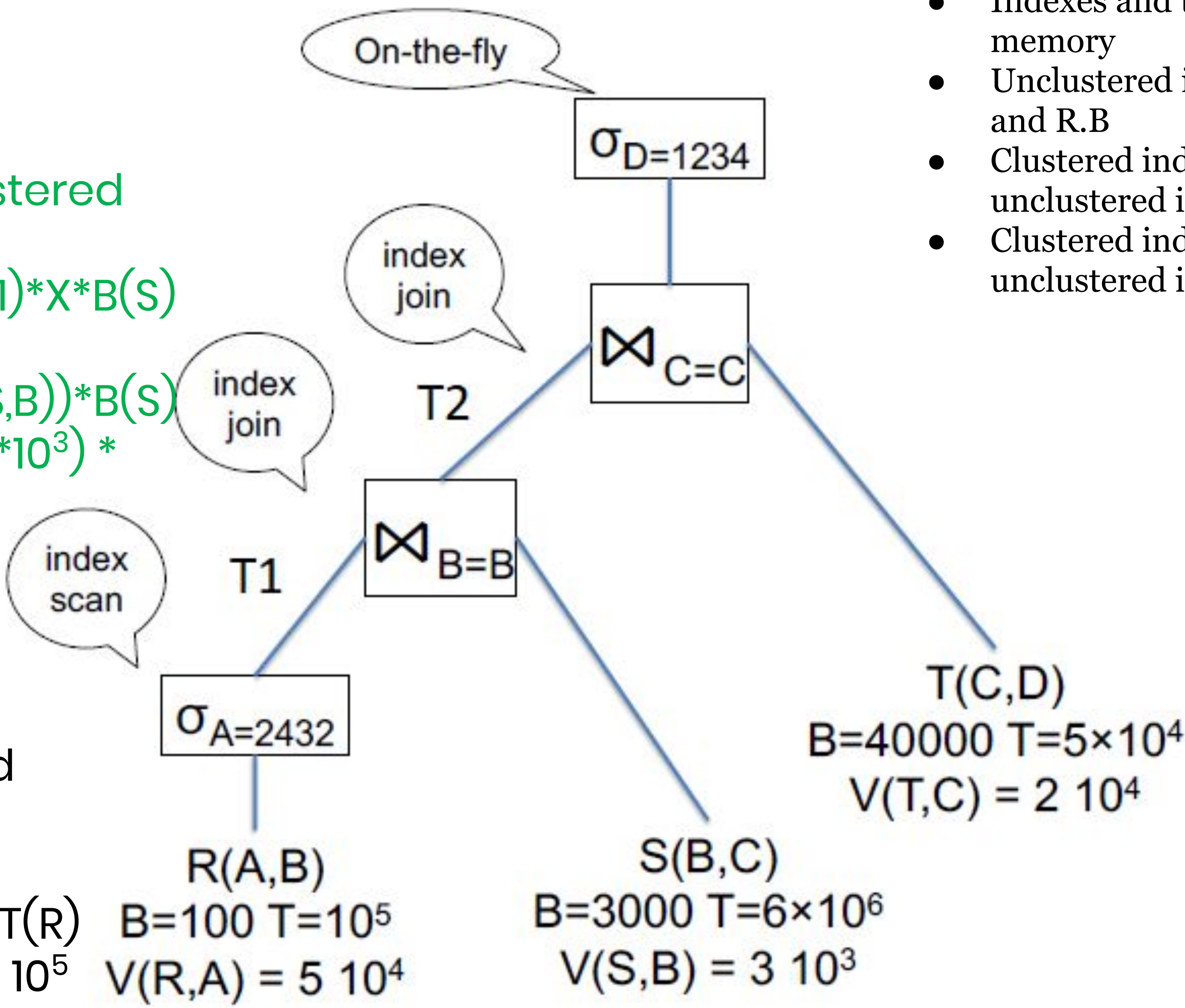
Assuming:

- Indexes and tuples fit in memory
- Unclustered indexes on R.A and R.B
- Clustered index on S.B, unclustered index on S.C.
- Clustered index on T.C, unclustered index on T.D.

Cost of clustered index join:
$= B(T1) + T(T1)*X*B(S)$
$= B(T1) +$
$T(T1) *(1/V(S,B))*B(S)$
$= 0 + 2*(1/ 3*10^3) *$
3000
$= 2$

$T(T1) = 2$

Cost of unclustered index scan:
$= X*T(R)$
$= (1/V(R,A))*T(R)$
$= (1/ 5*10^4) * 10^5$
$= 2$



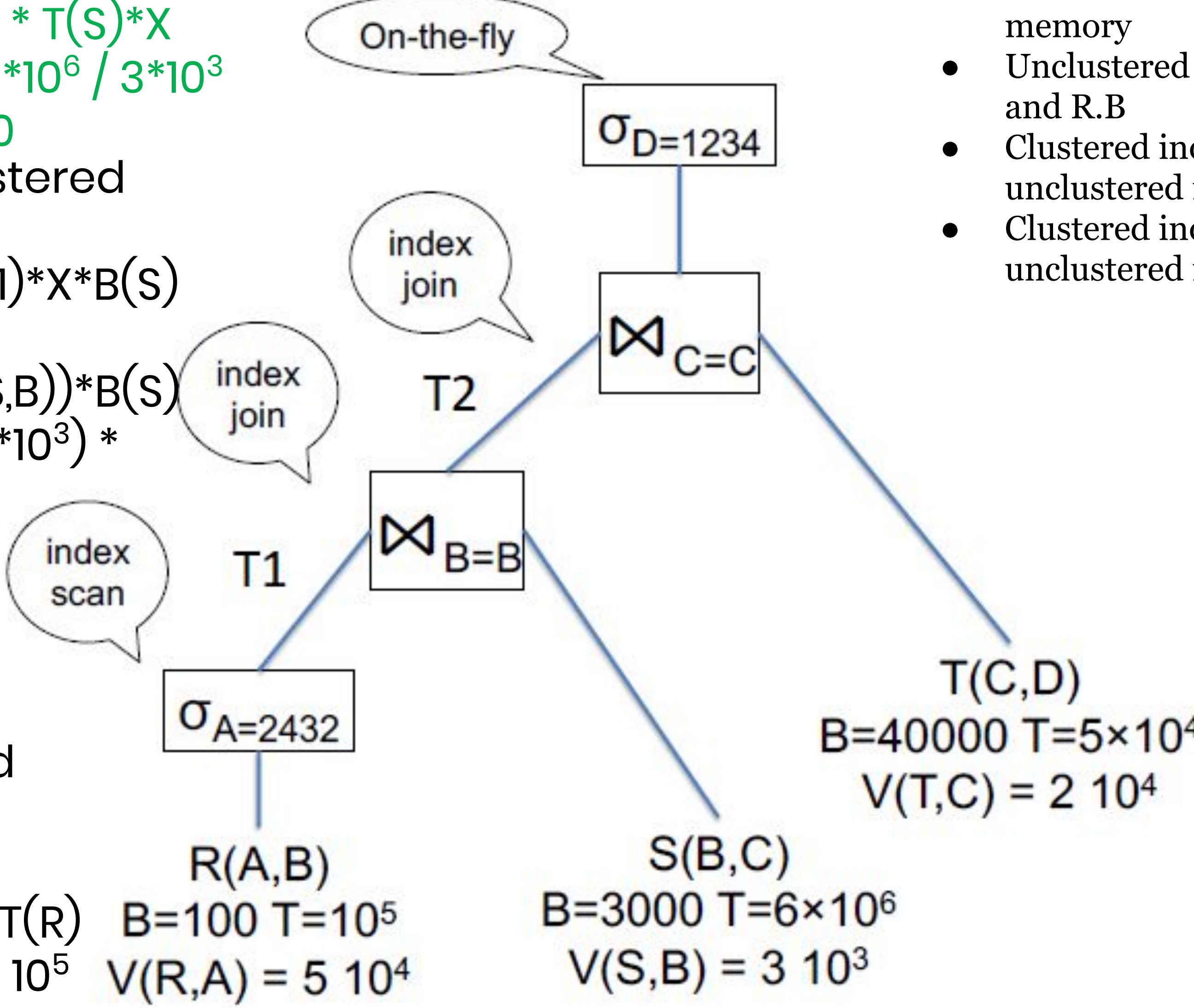
Example Problem

$T(T2)$
$= T(T1) * T(S)*X$
$= 2 * 6*10^6 / 3*10^3$
$= 4000$

Cost of clustered index join:
$= B(T1) + T(T1)*X*B(S)$
$= B(T1) +$
$T(T1) *(1/V(S,B))*B(S)$
$= 0 + 2*(1/ 3*10^3) *$
3000
$= 2$

$T(T1) = 2$

Cost of unclustered index scan:
$= X*T(R)$
$= (1/V(R,A))*T(R)$
$= (1/ 5*10^4) * 10^5$
$= 2$



Assuming:
- Indexes and tuples fit in memory
- Unclustered indexes on R.A and R.B
- Clustered index on S.B, unclustered index on S.C.
- Clustered index on T.C, unclustered index on T.D.

Example Problem

T(T2)
= T(T1) * T(S)*X
= 2 * $6*10^6$ / $3*10^3$
= 4000

Cost of clustered index join:
= B(T1) + T(T1)*X*B(S)
= B(T1) +
T(T1) *(1/V(S,B))*B(S)
= 0 + 2*(1/ $3*10^3$) *
3000
= 2

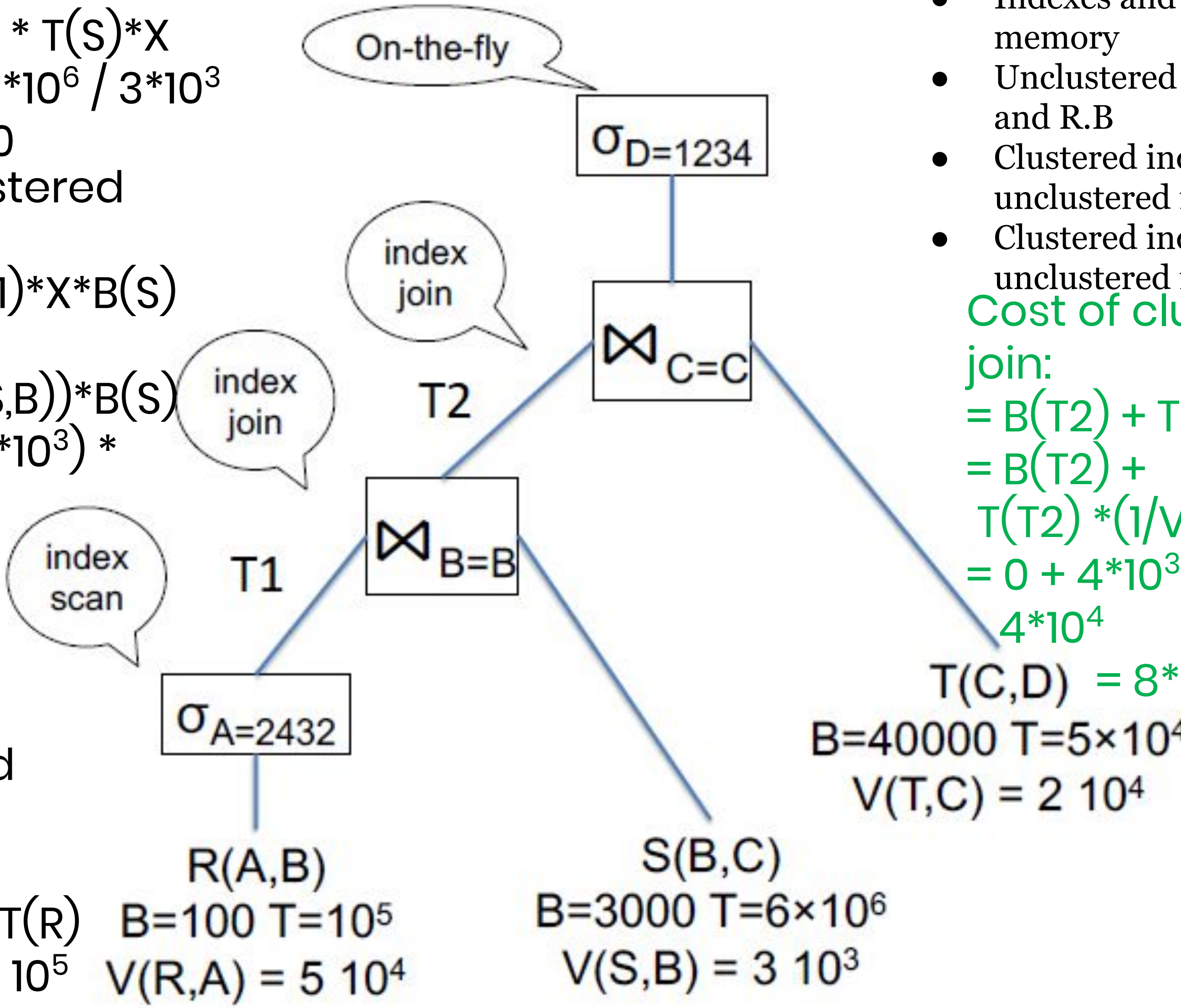
T(T1) = 2

Cost of unclustered index scan:
= X*T(R)
= (1/V(R,A))*T(R)
= (1/ $5*10^4$) * 10^5
= 2

Assuming:

- Indexes and tuples fit in memory
- Unclustered indexes on R.A and R.B
- Clustered index on S.B, unclustered index on S.C.
- Clustered index on T.C, unclustered index on T.D.

Cost of clustered index join:
= B(T2) + T(T2)*X*B(T)
= B(T2) +
T(T2) *(1/V(T,C))*B(T)
= 0 + $4*10^3$ *(1/ $2*10^4$) *
$4*10^4$
= $8*10^3$ = 8000



Example Problem

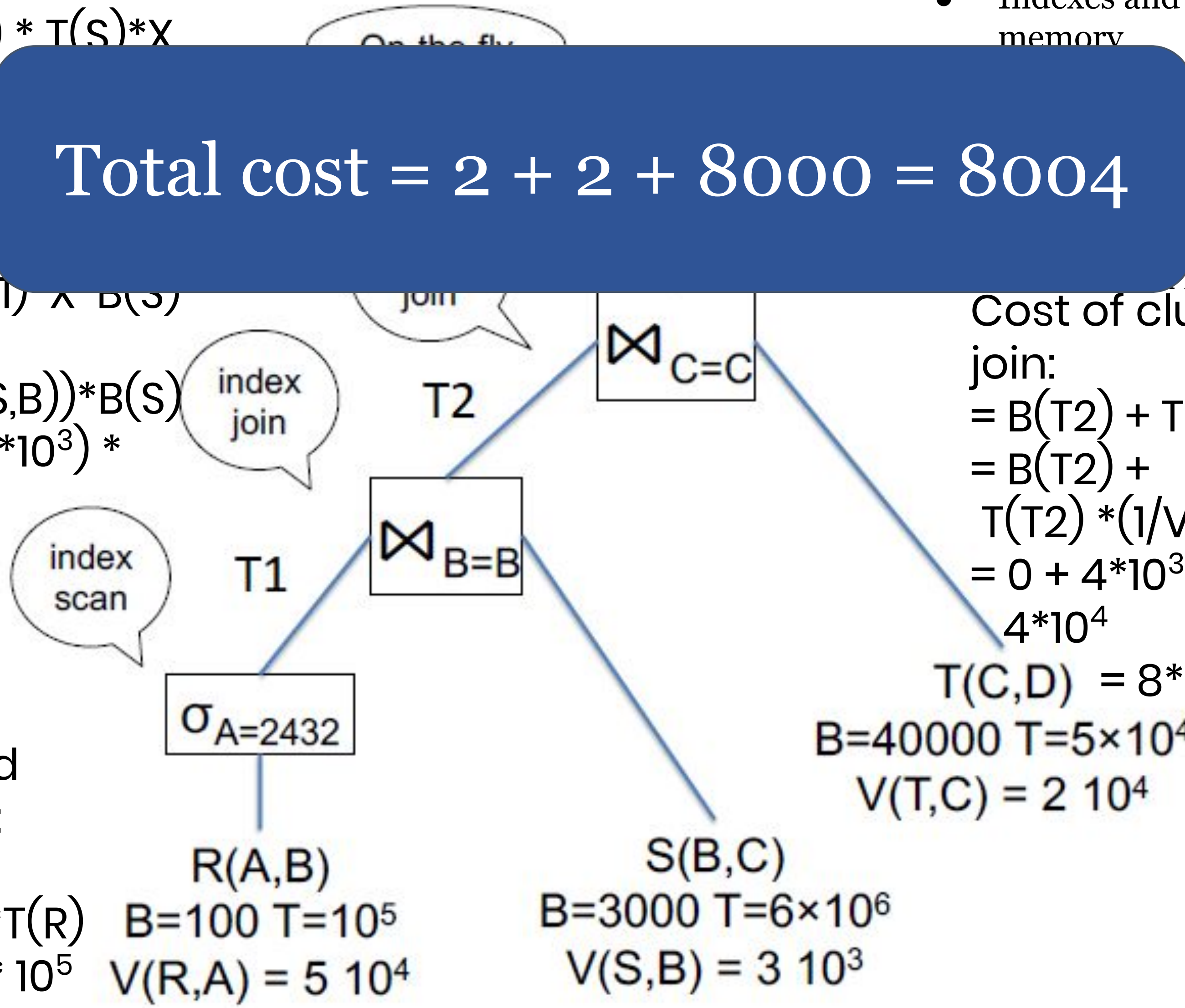
$T(T2)$
$= T(T1) * T(S)*X$
$= 2 * 6$
$= 400$

Cost of clus
index join:
$= B(T1) + T(T1)*X*B(S)$
$= B(T1) +$
$T(T1) *(1/V(S,B))*B(S)$
$= 0 + 2*(1/ 3*10^3) *$
3000
$=$ 2

$T(T1) = 2$

Cost of unclustered index scan:
$= X*T(R)$
$= (1/V(R,A))*T(R)$
$= (1/ 5*10^4) * 10^5$
$=$ 2

Total cost = 2 + 2 + 8000 = 8004



Assuming:
- Indexes and tuples fit in memory
- indexes on R.A
- lex on S.B,
- ndex on S.C.
- lex on T.C,
- index on T.D.

Cost of clustered index join:
$= B(T2) + T(T2)*X*B(T)$
$= B(T2) +$
$T(T2) *(1/V(T,C))*B(T)$
$= 0 + 4*10^3 *(1/ 2*10^4) *$
$4*10^4$

$T(C,D) = 8*10^3 =$ 8000