Module 12: Database Tuning

Winter 2026

Cheriton School of Computer Science

CS 348: Intro to Database Management

Reading Assignments and References

To be read during the Week of ???:

- Sections 12.1 through 12.4 of Chapter 12 of course textbook.[1]
- Sections 14.1 through 14.3, 14.6 and 14.7 of Chapter 14 of course textbook.
- Section 16.5 of Chapter 16 of course textbook.
- Section 25.1 of Chapter 25 of course textbook.

[1] Silberschatz, Korth and Sudarshan, *Database Systems Concepts*, 7^{th} edition

Outline

Database Tuning

The performance of a deployed information system may not satisfy requirements:

- the response time for some transactions may not be acceptable, or
- the throughput rate possible for some transactions may not be sufficient.

Database Tuning

The performance of a deployed information system may not satisfy requirements:

- the response time for some transactions may not be acceptable, or
- the throughput rate possible for some transactions may not be sufficient.

Database Tuning

Database tuning primarily entails adjusting *database parameters* and *physical database design* to address performance issues with transaction response time and throughput.

Database Tuning

The performance of a deployed information system may not satisfy requirements:

- the response time for some transactions may not be acceptable, or
- the throughput rate possible for some transactions may not be sufficient.

Database Tuning

Database tuning primarily entails adjusting *database parameters* and *physical database design* to address performance issues with transaction response time and throughput.

Choice of technology can also make it necessary to modify conceptual database design and application DML code to address such issues.

Database Tuning

The performance of a deployed information system may not satisfy requirements:

- the response time for some transactions may not be acceptable, or
- the throughput rate possible for some transactions may not be sufficient.

Database Tuning

Database tuning primarily entails adjusting *database parameters* and *physical database design* to address performance issues with transaction response time and throughput.

Choice of technology can also make it necessary to modify conceptual database design and application DML code to address such issues.

An appreciation of application workload and resource bottlenecks is crucial to choosing parameters and to physical design.

Database Tuning (cont'd)

Workload Description

A *workload description* contains:

- the critical queries and their frequency,
- the critical updates and their frequency, and
- the desired *performance goal* for each query or update.

For each query:

- which relations are accessed,
- which attributes are retrieved, and
- which attributes occur in selection/join conditions, and how *selective* are query conditions.

For each update:

- the type of update and relations/attributes affected, and
- which attributes occur in selection/join conditions, and how *selective* are query conditions.

Database Tuning (cont'd)

Performance goals:

- reducing the time needed to compute answers to a query, or
- transaction throughput for transactions entailing updates.

Database Tuning (cont'd)

Performance goals:

- reducing the time needed to compute answers to a query, or
- transaction throughput for transactions entailing updates.

Core issues: *How can a DBA group*

1. *make queries run faster* (via data structures, clustering, replication),
2. *make updates run faster* (via locality of data items), and
3. *minimize congestion due to concurrency.*

Outline

Physical Database Design

Recall the following from Module 10:

Standard Physical Design

A *standard physical design* for a relational schema defines the following for each relation name R:

1. a primary index for R *materializing* its extension as a concrete data structure, and
2. zero or more secondary indices for R materializing *projections* of the primary index for R as concrete data structures.

A materialization of a relation adds an additional *record identifier* (RID) attribute to the relation.

Secondary indices usually include the RID attribute of the primary index in their projection of R.

Physical Database Design

Recall the following from Module 10:

Standard Physical Design

A *standard physical design* for a relational schema defines the following for each relation name R:

1. a primary index for R *materializing* its extension as a concrete data structure, and
2. zero or more secondary indices for R materializing *projections* of the primary index for R as concrete data structures.

A materialization of a relation adds an additional *record identifier* (RID) attribute to the relation.

Secondary indices usually include the RID attribute of the primary index in their projection of R.

Primary and secondary indices are a variety of materialized views.

Materialized Views

Possible syntax in SQL:†

```
CREATE VIEW <view-name> [AS] (<query>)
MATERIALIZED [AS <data-structure-choice>]
```

†Declaring views to be materialized is not part of the SQL standard. Most RDBMSs also support an alternative "`CREATE INDEX ...`" syntax.

Materialized Views

Possible syntax in SQL:[†]

```
CREATE VIEW <view-name> [AS] (<query>)
MATERIALIZED [AS <data-structure-choice>]
```

Example (from Module 10): *Relation name*

```
PROF/(pnum,lname,dept)
```

with the standard physical design consisting of:

1. the Btree primary index on `pnum` called `PROF-PRIMARY`

```
create view PROF-PRIMARY as (select * from PROF)
materialized as BTREE with search key (pnum)
```

[†] Declaring views to be materialized is not part of the SQL standard. Most RDBMSs also support an alternative "`CREATE INDEX ...`" syntax.

Materialized Views

Possible syntax in SQL:†

```
CREATE VIEW <view-name> [AS] (<query>)
MATERIALIZED [AS <data-structure-choice>]
```

Example (from Module 10): *Relation name*

`PROF/(pnum,lname,dept)`

with the standard physical design consisting of:

1. the Btree primary index on `pnum` called `PROF-PRIMARY`

```
create view PROF-PRIMARY as (select * from PROF)
materialized as BTREE with search key (pnum)
```

2. and a Btree secondary index on `lname` called `PROF-SECONDARY`

```
create view PROF-SECONDARY
as (select lname, rid from PROF-PRIMARY)
materialized as BTREE with search key (lname)
```

---

†Declaring views to be materialized is not part of the SQL standard. Most RDBMSs also support an alternative "`CREATE INDEX ...`" syntax.

On Choice of Data Structure

A Btree data structure is just one of a number of possibilities an RDBMS might allow for `<data-structure-choice>`:

- ISAM or VSAM file (static ordered indices),
- an unsorted heap file,
- a hash file,
- R trees (and other multidimensional data structures),
- an array (for memory resident indices),
- ...

On Choice of Data Structure

A Btree data structure is just one of a number of possibilities an RDBMS might allow for `<data-structure-choice>`:

- ISAM or VSAM file (static ordered indices),
- an unsorted heap file,
- a hash file,
- R trees (and other multidimensional data structures),
- an array (for memory resident indices),
- ...

Workloads exists and are common that favour any of these.

$\Rightarrow$ *navigational applications and R trees*

$\Rightarrow$ *OLAP workloads, main memory arrays, and heap files*

On Choice of Data Structure

A Btree data structure is just one of a number of possibilities an RDBMS might allow for `<data-structure-choice>`:

- ISAM or VSAM file (static ordered indices),
- an unsorted heap file,
- a hash file,
- R trees (and other multidimensional data structures),
- an array (for memory resident indices),
- ...

Workloads exists and are common that favour any of these.

$\Rightarrow$ *navigational applications and R trees*

$\Rightarrow$ *OLAP workloads, main memory arrays, and heap files*

Records encoding tuples in an index can also be co-clustered with records encoding tuples for another index.

On Co-Clustering Indices

Co-Clustering

Two indices are *co-clustered* if the data pages for the indices are in common, that is, if records encoding tuples in the indices are interleaved within the same data pages.

On Co-Clustering Indices

Co-Clustering

Two indices are *co-clustered* if the data pages for the indices are in common, that is, if records encoding tuples in the indices are interleaved within the same data pages.

Co-clustering is useful when (1, N) relationships exist between the records for the two indices.

Example: *If a foreign key exists from an* `EMPLOYEE` *table to a* `DEPARTMENT` *table, each record for the primary index of* EMPLOYEE *might be co-clustered with its related* DEPARTMENT *record in the data pages of the* DEPARTMENT *primary index. Records for indices on other tables, e.g.,* `PROJECTS` *and* `JOBS` *can in turn be respectively co-clustered with* `DEPARTMENT` *and* `EMPLOYEE`*.*

On Co-Clustering Indices

Co-Clustering

Two indices are *co-clustered* if the data pages for the indices are in common, that is, if records encoding tuples in the indices are interleaved within the same data pages.

Co-clustering is useful when (1, N) relationships exist between the records for the two indices.

Example: *If a foreign key exists from an* `EMPLOYEE` *table to a* `DEPARTMENT` *table, each record for the primary index of* EMPLOYEE *might be co-clustered with its related* DEPARTMENT *record in the data pages of the* DEPARTMENT *primary index. Records for indices on other tables, e.g.,* `PROJECTS` *and* `JOBS` *can in turn be respectively co-clustered with* `DEPARTMENT` *and* `EMPLOYEE`*.*

Performance implications:

- Can speed up joins, particularly foreign-key joins; and
- Sequential scans of either relation become slower.

On Search in Ordered Indices

Ordered indices allow more general subqueries to be evaluated efficiently.

range queries

B-trees can also help evaluating a query with the form

```
select * from <table>
where <attribute> >= <constant>
```

If the search key of a Btree is on `<attribute>`, it can be used to efficiently locate tuples where `<attribute> = <constant>`. The remaining records that qualify are obtained by scanned the remaining data pages.

On Search in Ordered Indices (cont'd)

multi-attribute search keys

It is usually possible to create an index on several attributes of the same relation.

Example: *Assume relation name*

`PROF/(pnum,lname,fname,dept)`

with a secondary index now defined as follows:

```
create view PROF-SECONDARY
as (select lname, fname, rid from PROF-PRIMARY)
materialized as BTREE with search key (lname, fname)
```

Note that the order in which attributes appear in the search key is important.

In this index, records are organized first by `lname`*. Tuples with a common surname are then organized by* `fname`*.*

Using Multi-Attribute Indices

- *The* `PROF-SECONDARY` *index would be useful for these queries*:

```
select *
from PROF
where lname = 'Smith'
```

```
select *
from PROF
where lname = 'Smith'
and fname = 'John'
```

Using Multi-Attribute Indices

- *The* `PROF-SECONDARY` *index would be useful for these queries*:

```
select *
from PROF
where lname = 'Smith'
```

```
select *
from PROF
where lname = 'Smith'
and fname = 'John'
```

- *It would be very useful for these queries*:

```
select fname
from PROF
where lname = 'Smith'
```

```
select fname, lname
from PROF
```

Using Multi-Attribute Indices

- *The* `PROF-SECONDARY` *index would be useful for these queries*:

```
select *
from PROF
where lname = 'Smith'
```

```
select *
from PROF
where lname = 'Smith'
and fname = 'John'
```

- *It would be very useful for these queries*:

```
select fname
from PROF
where lname = 'Smith'
```

```
select fname, lname
from PROF
```

- *It is unlikely to be useful for this query*:

```
select *
from PROF
where fname = 'John'
```

Using Multi-Attribute Indices

- *The* `PROF-SECONDARY` *index would be useful for these queries*:

```
select *
from PROF
where lname = 'Smith'
```

```
select *
from PROF
where lname = 'Smith'
and fname = 'John'
```

- *It would be very useful for these queries*:

```
select fname
from PROF
where lname = 'Smith'
```

```
select fname, lname
from PROF
```

- *It is unlikely to be useful for this query*:

```
select *
from PROF
where fname = 'John'
```

Exercise: *Why would* `PROF-SECONDARY` *be very useful in the second case.*

Beyond Standard Physical Design

join indices

In standard physical design, a secondary index is a materialized view on a primary index.

Some systems support a materialized view defined by a conjunctive query on more than one primary index.

Example: *A join index for the bibliography database*:

```
create view AUTHOR-BOOK as (
    select author-rid, book-rid
    form AUTHOR-PRIMARY, WROTE-PRIMARY, BOOK-PRIMARY
    where aid = author and publication = pubid )
materialized as heap file
```

Beyond Standard Physical Design

join indices

In standard physical design, a secondary index is a materialized view on a primary index.

Some systems support a materialized view defined by a conjunctive query on more than one primary index.

Example: *A join index for the bibliography database*:

```
create view AUTHOR-BOOK as (
    select author-rid, book-rid
    form AUTHOR-PRIMARY, WROTE-PRIMARY, BOOK-PRIMARY
    where aid = author and publication = pubid )
materialized as heap file
```

arbitrary materialized views

Some systems support a materialized view defined by an arbitrary SQL query.

Query Optimization Revisited

View Based Query Rewriting

Given a query Q and a physical database design consisting of a set of arbitrary materialized views $\{V_1, \ldots, V_n\}$, the *view based query rewriting problem* is to find the most efficient query plan P equivalent to Q that uses only views V_i.

Query Optimization Revisited

View Based Query Rewriting

Given a query Q and a physical database design consisting of a set of arbitrary materialized views $\{V_1, \ldots, V_n\}$, the *view based query rewriting problem* is to find the most efficient query plan P equivalent to Q that uses only views V_i.

View based query rewriting is also a fundamental problem in *data integration* for which data sources are defined via the *local-as-view* paradigm.

Query Optimization Revisited

View Based Query Rewriting

Given a query Q and a physical database design consisting of a set of arbitrary materialized views $\{V_1, \ldots, V_n\}$, the *view based query rewriting problem* is to find the most efficient query plan P equivalent to Q that uses only views V_i.

View based query rewriting is also a fundamental problem in *data integration* for which data sources are defined via the *local-as-view* paradigm.

Theorem

View based query rewriting is undecidable, even when Q and each V_i is a conjunctive query and any plan P equivalent to Q is acceptable.

Query Plan Tools

An RDBMS will usually have a tool to enable one to investigate what plan is chosen for a query, and what the estimated cost is.

For DB2, one can use `db2expln` and `dynexpln`.

Example: *Invoking these tools on the bibliography query*

```
select name from author, wrote where aid = author
```

generates the following:

(default physical design)

```
Estimated Cost         = 50
Estimated Cardinality = 120

Optimizer Plan:

     RETURN
       |
     MSJOIN
    /      \
 TBSCAN   TBSCAN
   |        |
  SORT     SORT
   |        |
 TBSCAN   TBSCAN
   |        |
 Table:   Table:
 AUTHOR   WROTE
```

(secondary index `AW` on `author` attribute of `WROTE`)

```
Estimated Cost         = 25
Estimated Cardinality = 120

Optimizer Plan:

       RETURN
         |
       MSJOIN
      /      \
 TBSCAN      IXSCAN
   |        /      \
  SORT   Index:  Table:
   |     AW       WROTE
 TBSCAN
   |
 Table:
 AUTHOR
```

Index Advising Tools

An RDBMS will usually also have a tools to advise on a standard physical design given a workload description.

For DB2, one can use `db2advis`.

```
rees$ db2advis -d cs338
               -s "select name from author,wrote where aid=author"

Calculating initial cost (without recommmended indexes) [25.390385]
Initial set of proposed indexes is ready.
Found maximum set of [2] recommended indexes
Cost of workload with all indexes included [0.364030] timerons
total disk space needed for initial set [   0.014] MB
total disk space constrained to         [  -1.000] MB
  2  indexes in current solution
 [ 25.3904] timerons  (without indexes)
 [  0.3640] timerons  (with current solution)
 [%98.57] improvement

 Trying variations of the solution set.
--
-- execution finished at timestamp 2006-11-23-12.25.24.205770
--
-- LIST OF RECOMMENDED INDEXES
-- ===========================
-- index[1],    0.009MB
   CREATE INDEX WIZ8 ON "DAVID   "."AUTHOR" ("AID" ASC, "NAME" ASC) ;
-- index[2],    0.005MB
   CREATE INDEX AW ON "DAVID   "."WROTE" ("AUTHOR" ASC) ;
-- ===========================
Index Advisor tool is finished.
```

NOTE: `db2advis` can be given a more general workload.

Physical Design Guidelines

- Don't index unless the performance increase outweighs the update overhead.

Physical Design Guidelines

- Don't index unless the performance increase outweighs the update overhead.
- Attributes mentioned in WHERE clauses are candidates for index search keys.

Physical Design Guidelines

- Don't index unless the performance increase outweighs the update overhead.
- Attributes mentioned in WHERE clauses are candidates for index search keys.
- Multi-attribute search keys should be considered when:
  1. a WHERE clause contains several conditions; or
  2. it enables index-only plans.

Physical Design Guidelines

- Don't index unless the performance increase outweighs the update overhead.
- Attributes mentioned in WHERE clauses are candidates for index search keys.
- Multi-attribute search keys should be considered when:
  1. a WHERE clause contains several conditions; or
  2. it enables index-only plans.
- Choose indexes that benefit as many queries as possible.

Physical Design Guidelines

- Don't index unless the performance increase outweighs the update overhead.
- Attributes mentioned in WHERE clauses are candidates for index search keys.
- Multi-attribute search keys should be considered when:
  1. a WHERE clause contains several conditions; or
  2. it enables index-only plans.
- Choose indexes that benefit as many queries as possible.
- Each relation can have at most one primary index; therefore choose it wisely.
  1. Target important queries that would benefit the most.
     $\Rightarrow$ *range queries benefit the most from clustering*
     $\Rightarrow$ *join queries benefit the most from co-clustering*
  2. A multi-attribute index that enables an index-only plan does not benefit from being clustered.

Outline

Schema Tuning and Normal Forms

We have considered database tuning entirely from the perspective of physical database design. *But what if performance issues remain?*

Schema Tuning and Normal Forms

We have considered database tuning entirely from the perspective of physical database design. *But what if performance issues remain?*

It may be necessary to make changes to the conceptual database design and to *tune* queries.

Schema Tuning and Normal Forms

We have considered database tuning entirely from the perspective of physical database design. *But what if performance issues remain?*

It may be necessary to make changes to the conceptual database design and to *tune* queries.

Goals:

- avoiding expensive operations in query execution (joins), and
- retrieving *related data* in fewer operations.

Schema Tuning and Normal Forms

We have considered database tuning entirely from the perspective of physical database design. *But what if performance issues remain?*

It may be necessary to make changes to the conceptual database design and to *tune* queries.

Goals:

- avoiding expensive operations in query execution (joins), and
- retrieving *related data* in fewer operations.

Techniques:

- choosing an alternative normalization or weaker normal form;
- co-clustering relations (if supported) and denormalization;
- vertically and horizonally partitioning data and materialized views; and
- avoiding concurrency hot-spots.

Tuning the Conceptual Schema

Adjustments can be made to the conceptual schema:

re-normalization

Consider alternative BCNF decompositions that better fit the queries in the workload.

Tuning the Conceptual Schema

Adjustments can be made to the conceptual schema:

re-normalization

Consider alternative BCNF decompositions that better fit the queries in the workload.

In general, (re) normalization:

- speeds up simple updates (less change anomalies),
- speeds up simple queries (smaller tables),
- *slows down complex queries (more joins)*, and
- *slows down complex updates (if they involve complex queries)*.

Tuning the Conceptual Schema

Adjustments can be made to the conceptual schema:

re-normalization

Consider alternative BCNF decompositions that better fit the queries in the workload.

In general, (re) normalization:

- speeds up simple updates (less change anomalies),
- speeds up simple queries (smaller tables),
- *slows down complex queries (more joins)*, and
- *slows down complex updates (if they involve complex queries)*.

denormalization

Consider merging relational schemata to intentionally increase redundancy.

Tuning the Conceptual Schema

Adjustments can be made to the conceptual schema:

re-normalization

: Consider alternative BCNF decompositions that better fit the queries in the workload.

In general, (re) normalization:

- speeds up simple updates (less change anomalies),
- speeds up simple queries (smaller tables),
- *slows down complex queries (more joins)*, and
- *slows down complex updates (if they involve complex queries)*.

denormalization

: Consider merging relational schemata to intentionally increase redundancy.

In general, redundancy *increases update overhead* (due to change anomalies) but *decreases query overhead*.

Tuning the Conceptual Schema (cont'd)

Very large tables can be a source of performance bottlenecks.

Partitioning a table means splitting it into multiple tables for the purpose of reducing I/O cost or lock contention.

Tuning the Conceptual Schema (cont'd)

Very large tables can be a source of performance bottlenecks.

Partitioning a table means splitting it into multiple tables for the purpose of reducing I/O cost or lock contention.

horizontal partitioning

Each partition has all the original columns and a subset of the original rows.

Tuples are assigned to a partition based upon a (usually natural) criteria.

Often used to separate operational from archival data.

Tuning the Conceptual Schema (cont'd)

Very large tables can be a source of performance bottlenecks.

Partitioning a table means splitting it into multiple tables for the purpose of reducing I/O cost or lock contention.

horizontal partitioning

Each partition has all the original columns and a subset of the original rows.

Tuples are assigned to a partition based upon a (usually natural) criteria.

Often used to separate operational from archival data.

vertical partitioning

Each partition has a subset of the original columns and all the original rows.

Typically used to separate frequently-used columns from each other (concurrency *hot-spots*) or from infrequently-used columns.

Tuning Queries

Changes to the physical or conceptual schemas impacts *all* queries and updates in the workload.

It is sometimes desirable to target performance of specific queries or applications.

Tuning Queries

Changes to the physical or conceptual schemas impacts *all* queries and updates in the workload.

It is sometimes desirable to target performance of specific queries or applications.

Guidelines for tuning queries:

- Sorting is expensive. Avoid unnecessary uses of `ORDER BY`, `DISTINCT`, or `GROUP BY` clauses.

Tuning Queries

Changes to the physical or conceptual schemas impacts *all* queries and updates in the workload.

It is sometimes desirable to target performance of specific queries or applications.

Guidelines for tuning queries:

- Sorting is expensive. Avoid unnecessary uses of `ORDER BY`, `DISTINCT`, or `GROUP BY` clauses.
- It might be necessary to replace subqueries with joins.

Tuning Queries

Changes to the physical or conceptual schemas impacts *all* queries and updates in the workload.

It is sometimes desirable to target performance of specific queries or applications.

Guidelines for tuning queries:

- Sorting is expensive. Avoid unnecessary uses of `ORDER BY`, `DISTINCT`, or `GROUP BY` clauses.
- It might be necessary to replace subqueries with joins.
- It might be necessary to replace correlated subqueries with uncorrelated subqueries.

Tuning Queries

Changes to the physical or conceptual schemas impacts *all* queries and updates in the workload.

It is sometimes desirable to target performance of specific queries or applications.

Guidelines for tuning queries:

- Sorting is expensive. Avoid unnecessary uses of `ORDER BY`, `DISTINCT`, or `GROUP BY` clauses.
- It might be necessary to replace subqueries with joins.
- It might be necessary to replace correlated subqueries with uncorrelated subqueries.
- Use vendor-supplied tools to examine generated plans. Update and/or create statistics if poor plans are due to poor cost estimation.

Tuning Applications

Guidelines for tuning applications:

Tuning Applications

Guidelines for tuning applications:

- Minimize communication costs.
  1. Return the fewest columns and rows necessary.
  2. Update multiple rows with a WHERE clause rather than a cursor.

Tuning Applications

Guidelines for tuning applications:

- Minimize communication costs.
  1. Return the fewest columns and rows necessary.
  2. Update multiple rows with a WHERE clause rather than a cursor.
- Minimize lock contention and hot-spots.
  1. Delay updates as long as possible.
  2. Delay operations on hot-spots as long as possible.
  3. Shorten or split transactions as much as possible.
  4. Perform insertions/updates/deletions in batches.
  5. Consider lower isolation levels.

Summary

Physical database design has *enormous impact* on performance.

Summary

Physical database design has *enormous impact* on performance.

- Decisions require some *understanding* what the DBMS is doing.
  - $\Rightarrow$ *regarding query execution*
  - $\Rightarrow$ *regarding transaction processing*
  - $\Rightarrow$ *and regarding query optimization*

Summary

Physical database design has *enormous impact* on performance.

- Decisions require some *understanding* what the DBMS is doing.
  $\Rightarrow$ *regarding query execution*
  $\Rightarrow$ *regarding transaction processing*
  $\Rightarrow$ *and regarding query optimization*
- Modern systems provide many tools to assist.

Summary

Physical database design has *enormous impact* on performance.

- Decisions require some *understanding* what the DBMS is doing.
  - $\Rightarrow$ *regarding query execution*
  - $\Rightarrow$ *regarding transaction processing*
  - $\Rightarrow$ *and regarding query optimization*
- Modern systems provide many tools to assist.
- DBMS technology continues to be a very active area of research.