713. ETL Optimization

The ETL pathway (extraction, transformation & loading) can take up a notable amount of time for large datasets, which can sometimes lead to bottlenecks in the sense that e.g. a nightly loading of data is no longer possible. There are several possible optimizations which are not specific to i2b2, but have long been established in the classic data warehouse field.

Disabling Index Maintenance during Loading

As mentioned in the query optimization page, the i2b2 database schema contains several indexes to speed up query performance. When data is being loaded into indexed tables, the databases has to simultaneously update or extend both the actual table row data as well as the related index structures, in order to maintain index consistency at all times. Depending on the number of indexes this can incur a notable performance impact due to concurrent write operations across different areas of the physical hard drives. This effect becomes especially large when tables are not incrementally extended (like in a transactional setting), but truncated and fully loaded (like in a typical data warehouse setting).

It is therefore standard practice in data warehousing to disable or drop indexes before truncating and (re-)loading data into large tables. Even though the database will spend additional time rebuilding the indexes after loading, performance is improved as it is carried out sequentially and not simultaneously to table row data loading.

> **Exception**
>
> This rule does not apply if a table is being maintained incrementally in the sense of "delta updates". In this case indexes may in fact be beneficial to quickly locate rows to be updated

The following Oracle SQL script contains statements to drop indexes from the OBSERVATION_FACT table (part 1 - to be inserted into an ETL pathway before fact loading) and to recreate the same indexes (part 2 - to be inserted into an ETL pathway after fact loading).

```
-- 1. Drop OBSERVATION_FACT indexes
DROP INDEX of_ctx_blob;
DROP INDEX fact_nolob;
DROP INDEX fact_patcon_date_prvd_idx;
DROP INDEX idrt_fact_cnpt_pat_enct_idx;
DROP INDEX idrt_fact_mdf_pat_enct_idx;

-- 2. Recreate OBSERVATION_FACT indexes
CREATE INDEX of_ctx_blob                 ON observation_fact (observation_blob) indextype is ctxsys.context
parameters ('sync (on commit)');
CREATE INDEX fact_nolob                  ON observation_fact (patient_num, start_date, concept_cd,
encounter_num, instance_num, nval_num, tval_char, valtype_cd, modifier_cd, valueflag_cd, provider_id,
quantity_num, units_cd, end_date, location_cd, confidence_num, update_date, download_date, import_date,
sourcesystem_cd, upload_id) LOCAL;
CREATE INDEX fact_patcon_date_prvd_idx   ON observation_fact (patient_num, concept_cd, start_date, end_date,
encounter_num, instance_num, provider_id, nval_num, valtype_cd) LOCAL;
CREATE INDEX idrt_fact_cnpt_pat_enct_idx ON observation_fact (concept_cd, instance_num, patient_num,
encounter_num) LOCAL;
CREATE INDEX idrt_fact_mdf_pat_enct_idx  ON observation_fact (modifier_cd, instance_num, patient_num,
encounter_num) LOCAL;
```

> The script should be invoked with the credentials of the schema containing the OBSERVATION_FACT table
>
> Please note that the script already contains both the optimized indexes as well as the LOCAL option for partitioning, which may need to be removed depending on availability and use of the partitioning feature.

Extending the COMMIT interval

Modern relational databases typically provide transactions to bundle write operations (INSERT, UPDATE) which are temporarily collected and then stored together when a "COMMIT" command is issued. Besides ensuring data consistency, this approach also allows the database to more efficiently store larger amounts of data at one time when compared to individually writing rows onto the disk.

Unless explicitly configured, database sessions often run in the so-called "AUTOCOMMIT" mode, which immediately write every single INSERT or UPDATE statement onto the disk. Raising the number of statements before an automatic commit is issued can significantly speed up data writing. }

> Please note that the database will store all statements running up to the next commit in a temporary area (e.g. the TEMP tablespace), which needs to be adequately sized to avoid tablespace overflows.

Using a dedicated bulk loader

Many databases provide a dedicated "bulk loader" tool which is optimized for loading large amounts of source data "en bloc" into a table, e.g. Oracle SQL*Loader. Such loaders often provide additional performance boosts through bypassing transaction- and consistency mechanisms (e.g. "DIRECT PATH" and "UNRECOVERABLE" options), which effectively stream data directly onto the disk. In a data warehouse setting, these kinds of optimizations are generally considered safe to use as the database is not used in a transactional setting and write operations are centrally controlled by an ETL pathway. Use of a bulk loader with enabled optimizations can lead to notable performance improvements of up to 100x when compared to individual INSERT statements.

Bulk loaders usually require CSV source files containing the raw data to be loaded and a separate control file containing all required information about the destination table and columns as well as optimization options to be applied.

ETL tools often provide access to bulk loaders through dedicated nodes or tasks. E.g. Talend Open Studio provides an insert node as well as a bulk loader node which are interchangeable.

> It should be noted that bulk loaders are database specific. If compatibility across several databases is required, it may be necessary to either create several adapted versions of the ETL pathway (affecting only the "final" step of actually loading generated CSV files) or to not use a bulk loader. E.g., the IDRT pathways stick to JDBC components to achieve greater cross-database compatibility)

Partially Loading Fact Data

Truncating and fully reloading fact table data is standard practice in data warehousing. However, for large tables this approach may not be feasible e.g. due to loading duration exceeding the available downtime of the warehouse.

Delta updates are an alternative approach which relies on timestamps or update flags in the source data to selectively insert new rows or update existing rows in the destination table. Depending on the volume of data changed between loading intervals this approach can notably reduce loading duration. However, a reliable source of timestamps/flags is required, which may be difficult to prove (is the source application reliably maintaining all creation or update timestamps with all possible transactions?). Alternatively, trigger-based mechanisms are possible to reliably carry over INSERTs or UPDATEs from a source system into the data warehouse. Delta updates are highly specific to the local setting and relevant source data, so their implementation details are out of scope of this guide.

Especially during the design phase or when new data types are being added to an i2b2 warehouse, it may be necessary to frequently reload only a specific portion of the overall dataset (e.g. only lab data). If the OBSERVATION_FACT table has been partitioned (as described in the page on query optimization), individual partitions can be quickly truncated, which can be notably faster than issuing DELETE statements for individual records or wildcarded concept codes.

The following Oracle SQL statement selectively truncates the LAB partition (laboratory findings):

```
ALTER TABLE observation_fact TRUNCATE PARTITION lab;
```

> The script should be invoked with the credentials of the schema containing the OBSERVATION_FACT table.
>
> Please note that the statement will delete all data in the LAB partition.