Financial Reporting Data Warehousing with CQL

Jee Chung, Ryan Wisnesky
Conexus AI

Abstract

Using a simplified financial reporting example, in this paper we examine how traditional data warehouses are put together and how they often fail due to challenges that are difficult to predict prior to the project's start and difficult to analyze afterwards. We then propose an improved method for creating data warehouses: using the categorical query language CQL. Finally, we demonstrate how CQL enables the early identification of, and solution to, a large class of common data warehousing problems, reducing the risk of failure and increasing project implementation speed.

Contents

1 Introduction **1**

2 How Data Warehousing is Done Today **2**
2.1 What Went Wrong? 3

3 Doing Data Warehousing the Forward (Right) Way **4**
3.1 Adding Value With Automated Theorem Proving 5

4 An Example Financial Reporting Warehouse In CQL **5**
4.1 The Data Sources 6
4.2 Schema Integration 6
4.3 Data Integration 7
4.4 Migrating to the Target Schema 8

5 Conclusion **8**

6 Appendix: Provenance **12**

7 Appendix: Trickle Feed / Incremental Load **13**

---

1 Introduction

While there are many IT systems for interacting with large volumes of data, the role of a traditional data warehouse is still very pertinent, especially when it comes to data that must be exact; for example, a data warehouse that a financial institution might use for client reporting. Consumers of these reports (clients) care about every line item and every number being exactly right. In some situations, there are laws governing data accuracy, privacy, and provenance. Such requirements demand that a data warehouse intended for such purposes be built with precision, accuracy, and verifiable correctness as the most important qualities. This is a hard problem for any data team to solve, and it is especially challenging for a team working to build a data warehouse by integrating heterogeneous data sources. Yet, that is the most common use case for data warehousing projects. It is thus no surprise

that traditional data warehousing projects are almost always over budget and more than half of them are failures. In most cases, hindsight shows that many classes of the problems encountered were predictable:

- Missing data (e.g., NULLs)
- Schema and data mismatches (e.g., Pete vs Peter)
- Problems uncovered too late to fix easily (e.g., after warehouse construction)
- Data integrity assumptions fail (e.g., social security number not a primary key)

In this paper we:

- Demonstrate that traditional data warehousing projects fail because they are implemented backwards, and
- Propose a forwards approach using CQL to make data warehousing projects more predictable, more accurate, faster to implement, and in a way that requires fewer design decisions – and therefore, less risky.

The example in the paper is built-in to the CQL IDE as "Finance Colim 1".

2 How Data Warehousing is Done Today

A typical financial reporting data warehousing project is implemented as follows:

1. A business sponsor identifies a business need to create a client reporting system.
2. The sponsor works with business analysts to identify what should be in the reports.
3. The project is formally identified, and IT is pulled in.
4. The IT team studies the results of the business analysis and creates the schema for a data warehouse that will meet the business requirements. It will have tables to represent clients, their accounts, their holdings, the portfolios their investments belong to, trade histories, etc.
5. The combined teams go "shopping" for the necessary data pieces. The result of this shopping spree is an inventory, a "data dictionary", of the data pieces from various sources within the firm. Data will likely come from systems such as the CRM system, the accounting system, the trading system, the general ledger system, various spreadsheets, and so on. At this point, only subjective assessments about the quality of the data exist and little is known about how well the data will fit together to populate the data warehouse.
6. The IT staff puts together a team of ETL developers who design a target schema that will support the requirements and the ETL development effort begins. At some point, weeks and perhaps months into the effort, data starts to come together in the target data warehouse. It becomes evident that there are challenges with missing data, poor data match, etc. In addition, constant scope-creep tends to make the initial target schema less and less valid. But because there has been so much invested already, and because the end business goal is firm, the tendency is to somehow make the data fit the target schema at the expense of quality and accuracy, making a bad situation worse. Over time, complications compound while the team continues their development effort.
7. Eventually, the effort has suffered so much rework and error-patching that it has either taken much longer than originally planned and/or the project is deemed a failure and is shut down.

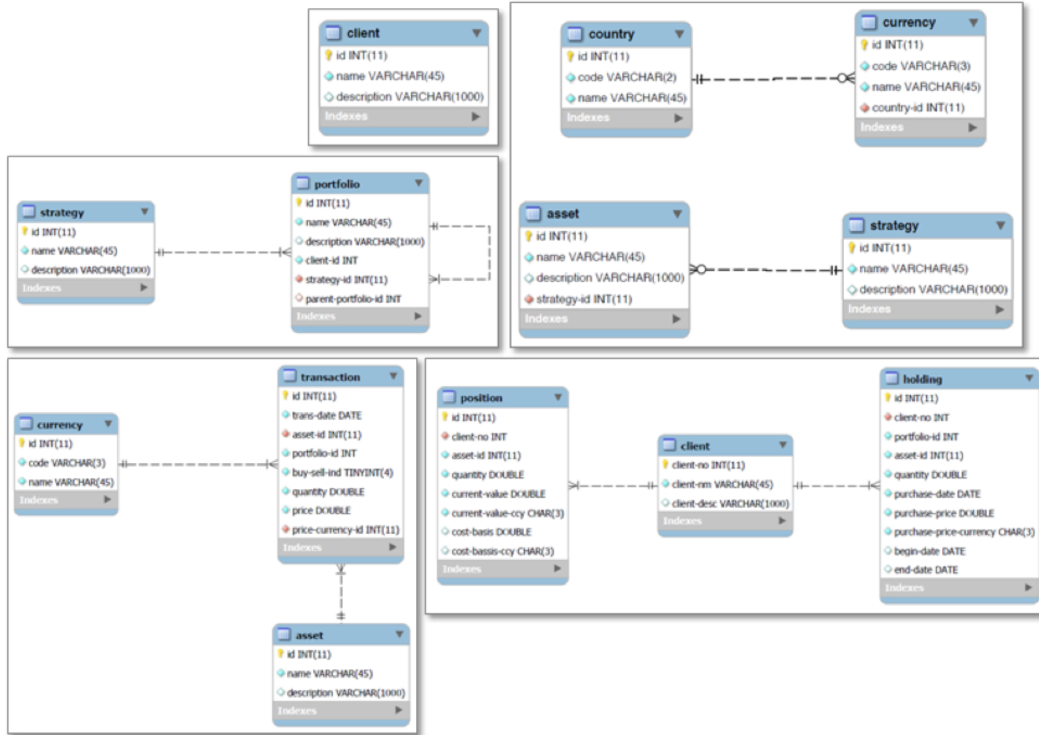

Figure 1: An example of data sources based on "shopping" for data ingredients

Figure 2: The Traditional data warehousing approach is brittle

2.1 What Went Wrong?

There are usually multiple factors that contribute to data warehousing failure, and the signs of risk and challenges often appear early and are manageable if corrective measures are also taken early. However, the traditional data warehousing approach makes the early identification of risk nearly impossible, because users typically cannot find out what can go wrong during data integration until they actually integrate data. That integration effort involves an army of ETL developers trying to put the data together to meet the expected requirements (target schema) that were created prior to the full knowledge of the available data. The traditional warehousing process is done backwards, because it starts with reporting requirements which in turn drive the warehouse schema design, which may or may not be constructible by the available data.

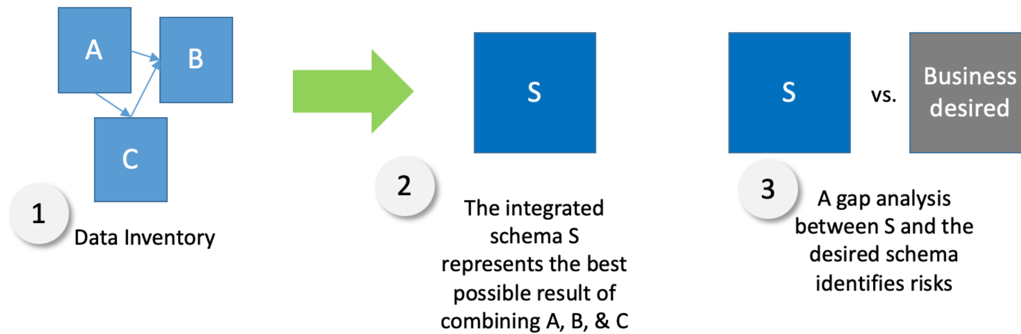

Figure 3: The "forward" approach allows early identification of data risks

3 Doing Data Warehousing the Forward (Right) Way

Learning from the mistakes of the backwards methodology, the CQL approach implements the same reporting data warehouse in a forward way, as a function of the source schemas rather than the target. The CQL approach starts like the backwards example:

1. A business sponsor identifies a business need to create a client reporting system.
2. The sponsor works with business analysts to identify what should be on the reports.
3. The project is formally identified, and IT is pulled in.
4. The IT team studies the results of the business analysis and creates the schema for a data warehouse that will meet the business requirements.
5. The combined team go "shopping" for the necessary data pieces, resulting in a "data dictionary".

... but here is where it starts to be different:

6. The IT team takes the data dictionary and uses CQL to define the relationships between the input schemas. From this specification, CQL computes the unique optimally integrated schema that takes into account all of schema matches, constraints, etc.
7. The integrated schema is compared to the desired data warehouse schema to create a gap analysis that measures the difference between the available source data and what the project team needs. Many of the data issues are uncovered before any ETL development begins and before any data is moved, allowing up-front risk mitigation.
8. The next step is similar to step 6, but at the data level: the IT team defines the record linkages (e.g. match by primary key) between elements related across input databases, and CQL emits the unique optimally integrated data warehouse compatible with the links. This step can be performed in memory using the CQL IDE itself, or performed via code generation to back-ends such as Spark and Hadoop.
9. The final step is to use the queries and schema mappings from step 7 to migrate data from the integrated schema onto the given desired target data warehouse schema.

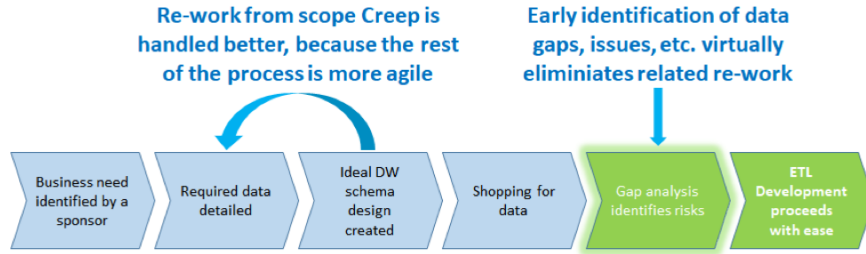

Figure 4: Data warehousing the "forwards" way identifies risks early

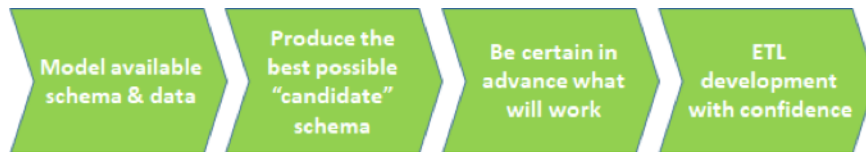

Figure 5: Data warehousing the CQL Way

3.1 Adding Value With Automated Theorem Proving

All of the processes described above are expedited by the CQL automated theorem prover, which automatically checks that schema mappings and queries preserve data integrity constraints across all possible input databases. As a result, power users can write queries that would be otherwise impossible to get correct, and domain experts without IT training can safely employ a "guess and check" program development strategy with 100% confidence that their CQL programs cannot violate data integrity. To summarize, CQL performs data integration with mathematical precision and predictability so that one can execute data warehousing project with confidence that:

- Data risk are identified early, at compile/design time, by the CQL automated theorem prover, and
- Data integration is mathematically optimal and as such the results have the highest quality required to meet the needs of projects like financial reporting.

Moreover, CQL allows a data warehousing project to proceed in a more natural and deterministic manner, thereby reducing development time and design risk:

1. Start with high quality model of available data sources.
2. Produce a mathematically optimized integrated model.
3. Compare the above to the desired warehouse–know in advance what is possible and what issues there are.
4. Execute data integration with confidence.

We now examine the financial client reporting data warehouse example in detail.

4 An Example Financial Reporting Warehouse In CQL

The following example is built-in to the CQL IDE as "Finance Colim 1". Our example warehousing scenario proceeds as follows:

1. Model the data sources in CQL.

2. Provide CQL with information about how the source schemas are related. CQL provides compile-time validation.
3. CQL computes the integrated schema. At this point, an offline gap analysis can determine how to map the integrated schema to the target schema. Again, CQL provides compile-time validation.
4. Model the desired target schema (what the business needs) in CQL.
5. Provide CQL with information about how the source data are related. CQL provides both compile-time and run-time validation.
6. Deploy the CQL program onto the desired back end, or execute it on the local machine.

As the project requirements evolve, the above steps can be repeated in an iterative manner. Note that CQL has modularized development in such a way that requirement changes to the target schema will not affect the integrated schema, which is purely a function of the source schemas.

4.1 The Data Sources

The following databases make up our financial reporting example and are shown in Figure 6:

- Client - a database with a list of clients.
- Reference - a database with reference lists of countries, currencies, investment strategies, and investable assets.
- Portfolio - a database with list of client investment portfolios; includes a copy of the strategies as found in the "Reference" database above.
- Transaction - a database with records of trades.
- HoldPos - a database with records of holdings and positions for the portfolios.

We have included some complications in the example databases to make them more realistic:

- The portfolio database contains a copy of the strategy table, a common practice in organizations that have disparate systems whose reference data must be coordinated.
- Likewise, the transaction database contains replicas of the asset and currency tables, and the holding-and-position databases contains a copy of the client table, but with the names of the columns slightly different from the copy in the client database.
- The holding and position tables use currency code values instead of currency code IDs.

As we go through the CQL-based data integration, we will examine how these complications, as well as the basic characteristics of the source databases, are handled by CQL.

4.2 Schema Integration

In CQL, one way to compute the integrated schema is by using the mathematical operations of *disjoint union* and *quotient*. This process takes as input equations relating the tables and columns of the input schemas. CQL code that does this in our example is shown in Figure 7. Figure 9 shows a visual representation of the integrated schema. Note that in this example:

- Client, Ref, Trans, Portfolio, and HoldPos are names of schemas of the data sources.
- The entity equations section defines which tables are related.

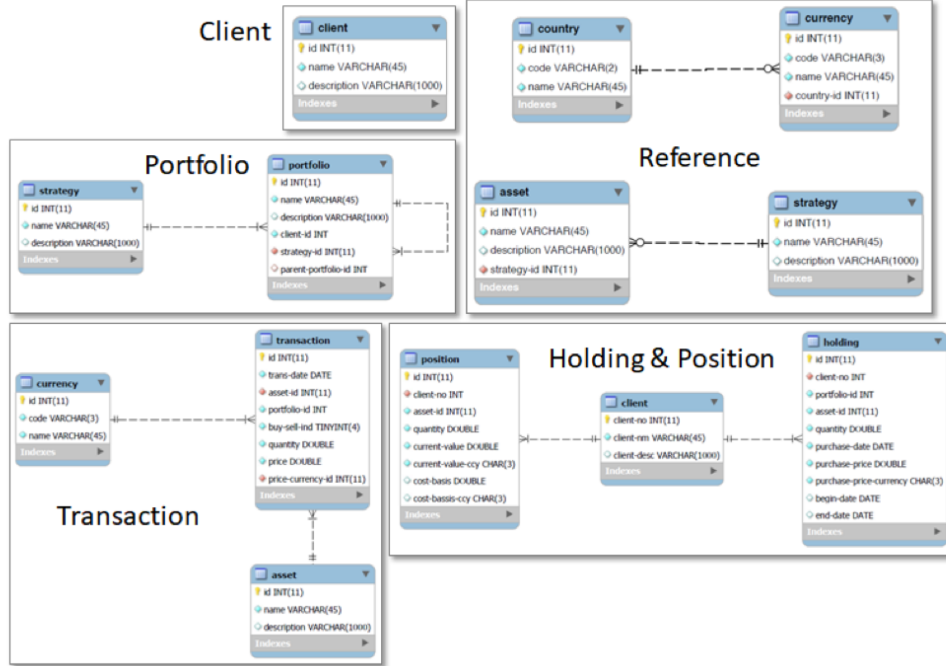

Figure 6: Source databases for our example

- The observation equations section defines which columns are related.
- The result of the quotient operation is the schema named Colimit. The name Colimit comes from the fact that in CQL, integrated schemas are *co-limits* in the sense of category theory, the branch of mathematics from which CQL, the categorical query language, derives its name.

It is also worth noting that, unlike traditional ETL tools, the relations between columns in CQL need not be simple correspondences, but can involve arbitrary equationally specified computations. For example, it is possible to relate an employee's age to their salary using a (e.g., Excel) formula, which will be taken into account by CQL's automated theorem prover during schema integration and during migration from the integrated schema to the target warehouse schema.

4.3 Data Integration

The first step of the data integration process is to migrate each data source onto the integrated schema individually. This step is fully automated using CQLs *sigma* operation, and the resulting disjoint union of databases will be merged in the second step. In some cases, merging may not even be necessary. In other cases, merging must be performed according to black-box rules about how data is related. For this reason, CQL reduces the problem of data integration to the problem of specifying relations between rows, in a way analogous to how CQL reduces the problem of schema integration to the problem of specifying relations between columns: mathematically, both operations are quotients. In this example, we match related elements based on their primary keys, as expressed in a CQL query written in SQL-ish notation, part of which reads:

```
FROM a b : Client__HoldPos WHERE a.Client_client_id = b.HoldPos_client_no
```

indicating that clients and holding positions are related by ID and client number fields. The overall data integration process is shown in Figure 8.

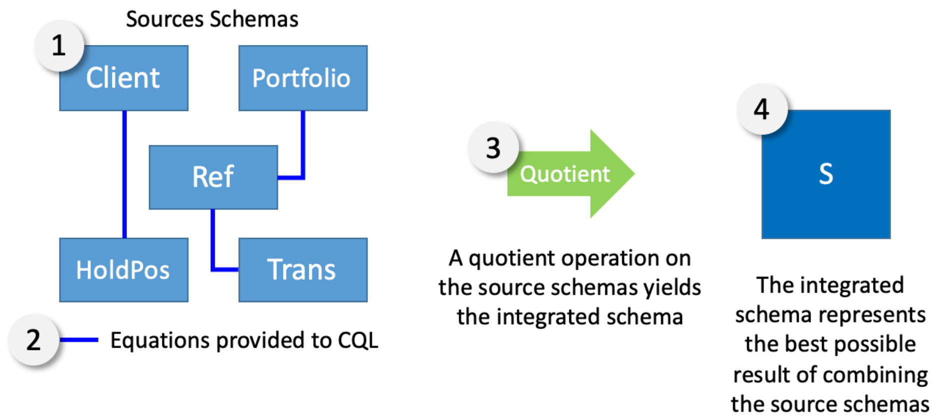

Figure 7: Schema Integration Using Quotients

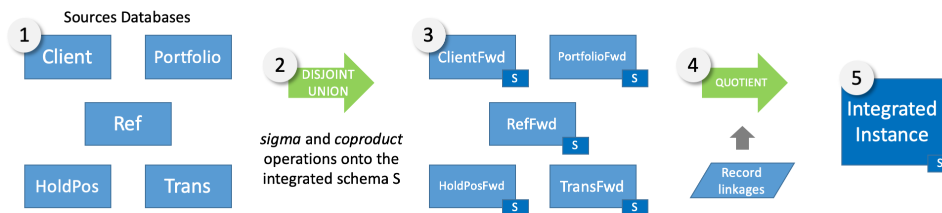

Figure 8: Data Integration Using Quotients

4.4 Migrating to the Target Schema

Finally, the target schema induced by the business requirements for the data warehouse is populated with confidence that the involved mappings and queries are correct, because they were already validated during gap analysis after the construction of the integrated schema. In our example, the target warehouse schema has the familiar look of a star schema design that is conducive to efficient reporting. To map the integrated schema to the target we define a query, a kind of generalized schema mapping written in SQL-ish notation, shown in Figure 15. An example of how the CQL IDE interactively enforces correctness of queries is shown in Figure 13.

5 Conclusion

Traditional data warehouse projects have typically used the approach of starting with business requirements and then trying to build a data warehouse that meets the requirements without validating a priori that the requisite data even exists. This process often results in unmet requirements that cannot be discovered until well into the implementation effort, resulting in costly rework or failure. Although the idea of creating a data warehouse with knowledge of what data can be delivered guiding the design of the integrated schema may not be new, until CQL it was not possible to deliver on this idea with mathematical certainty and precision. CQL is not a database and it is not a server. It is a functional programming language and a dedicated IDE for ETL tasks. It provides the means to execute all the data integration tasks necessary to create a precision data warehouse, where critical schema and data mappings are validated by the rigor of mathematics at both compile time and run time, thereby reducing risk, improving implementation speed and reducing cost.

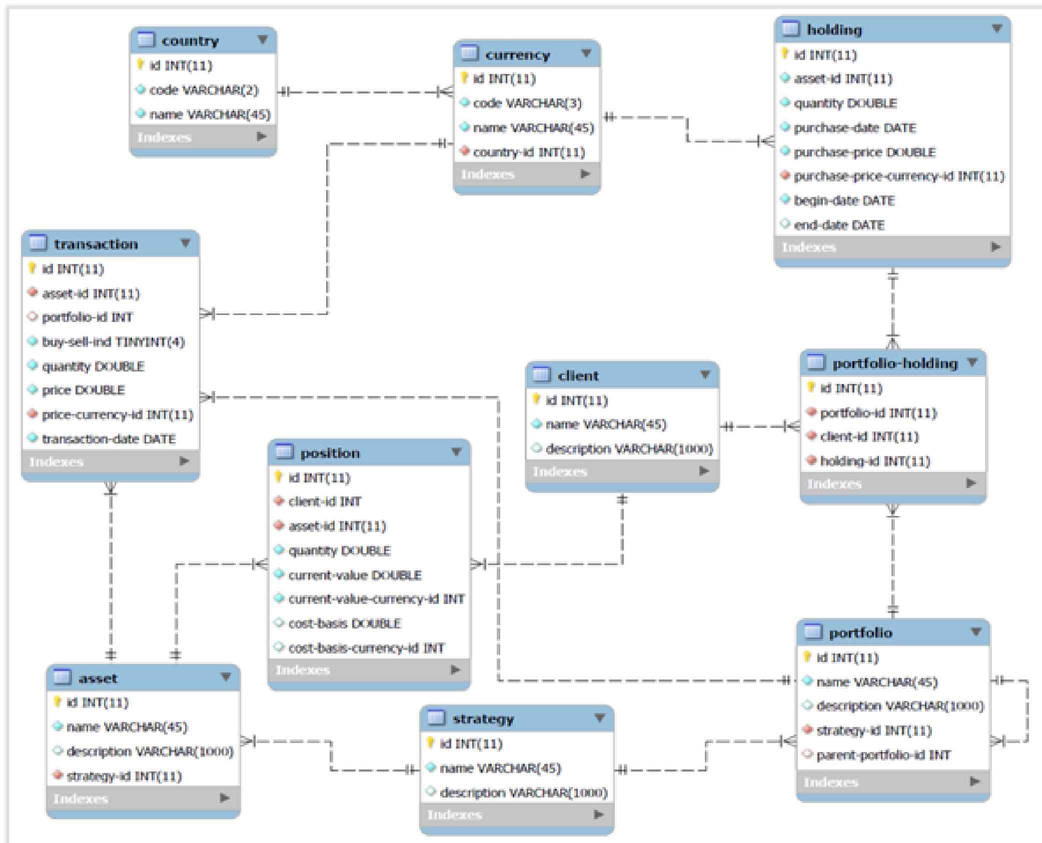

Figure 9: A visual representation of CQL-computed colimit schema

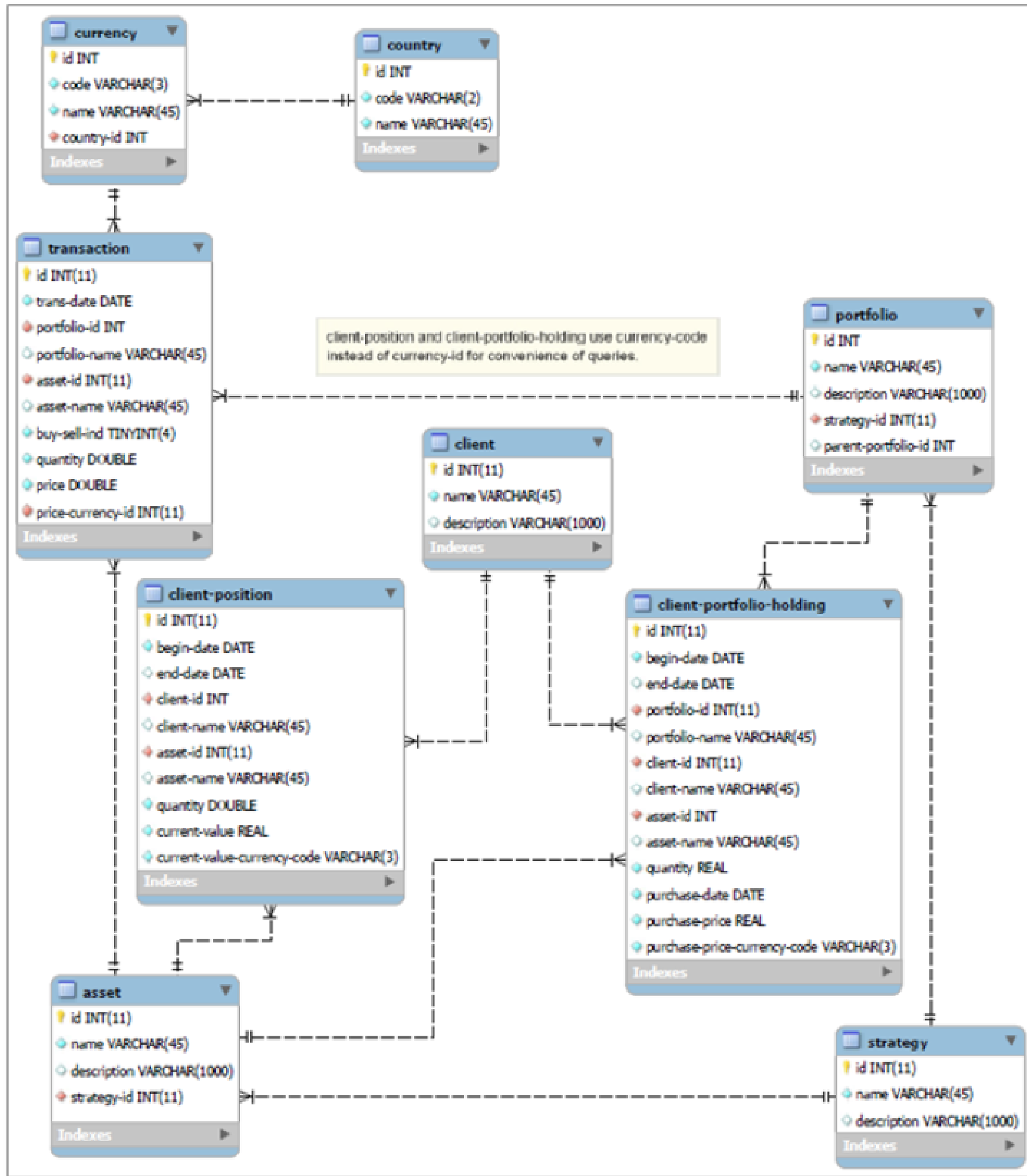

Figure 10: Target Star Schema to be projected from the integrated schema

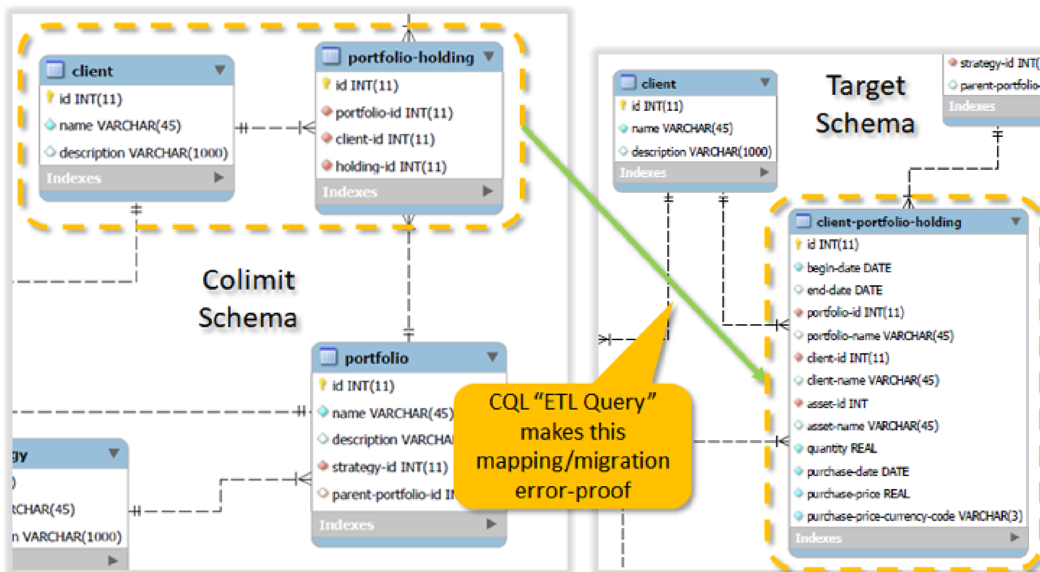

Figure 11: Mapping client and portfolio-holding entities from the Integrated schema to the target schema

Target schema (shown abridged here)

```
schema FinTarget = literal : Ty {
  entities ...
  foreign_keys ...
  path_equations ...
  attributes ...
  observation_equations ...
}
```

ETL query block

ETL Query for client-portfolio-holding

```
query ETL = literal : Colimit -> FinTarget {

   ...

  entity       clientportfolioholding -> {
    from ph:portfolioholding
    attributes clientportfolioholding_id -> ph.portfolioholding_holding.holding_id
      clientportfolioholding_begin_date -> ph.portfolioholding_holding.holding_begin_date
      clientportfolioholding_end_date -> ph.portfolioholding_holding.holding_end_date
      clientportfolioholding_asset_asset_id -> ph.portfolioholding_holding.holding_asset.asset_id
      clientportfolioholding_asset_asset_name -> ph.portfolioholding_holding.holding_asset.asset_name
      clientportfolioholding_purchase_price -> ph.portfolioholding_holding.holding_purchase_price
      clientportfolioholding_quantity -> ph.portfolioholding_holding.holding_quantity
      clientportfolioholding_purchase_date -> ph.portfolioholding_holding.holding_purchase_date
      clientportfolioholding_client_client_id -> ph.portfolioholding_client.client_id
      clientportfolioholding_client_client_name -> ph.portfolioholding_client.client_name
      clientportfolioholding_portfolio_portfolio_id -> ph.portfolioholding_portfolio.portfolio_id
      clientportfolioholding_portfolio_portfolio_name -> ph.portfolioholding_portfolio.portfolio_name
      clientportfolioholding_currency_currency_code -> ph.portfolioholding_holding.holding_currency.currency_code
    foreign_keys
      clientportfolioholding_client -> {c -> ph.portfolioholding_client}
      clientportfolioholding_asset -> {a -> ph.portfolioholding_holding.holding_asset}
      clientportfolioholding_portfolio -> {p -> ph.portfolioholding_portfolio}
  }

  ...

}
```

Figure 12: Example CQL ETL Query

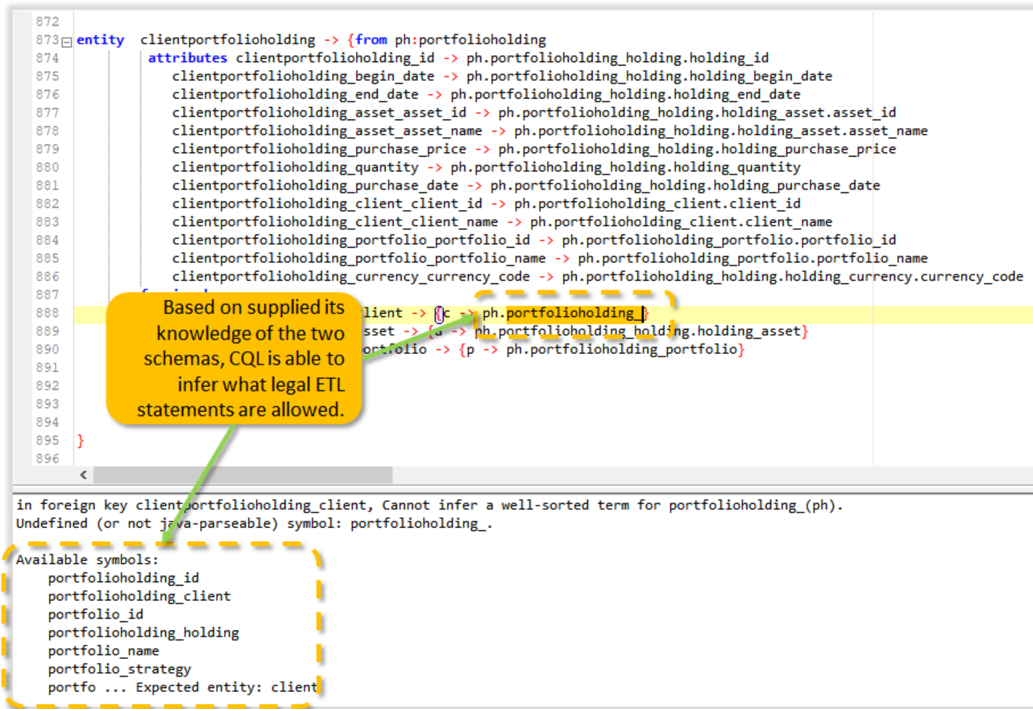

Figure 13: CQL IDE enforces code-time validation of mappings

6 Appendix: Provenance

The CQL IDE maintains the full provenance of every computation it executes. In this section, we describe an example of a particular subset of that provenance; namely, a simple form a lineage associated to each output record. We begin by showing the final result of our query, which contains row numbers, and also show the same result, with the "provenance box" checked; with this box checked, we can see that the row numbers are replaced by machine-interpretable information; this information will refer to the provenance and row identifiers of other, previous computations:

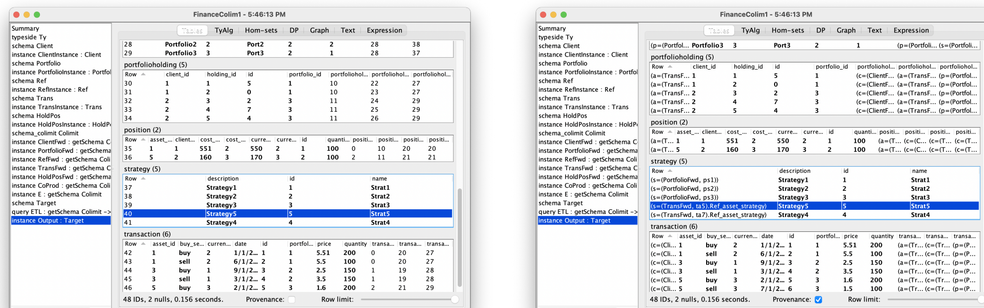

Figure 14: Example Provenance in Output

To see how this works, consider the "strategy 5" highlighted in the above window. Its associated lineage mentions "s", referring the the FROM-bound variable in the query that created it, along with "(TransFwd, ta5).Ref-asset-strategy". If we refer back to the "TransFwd" database, one of the intermediate results in our ETL flow, we see a cell that can be uniquely identified using the lineage of the output tuple, and it is highlighted in the diagram below; moreover, the "ta5" identifier can be traced back all the way to the original data:

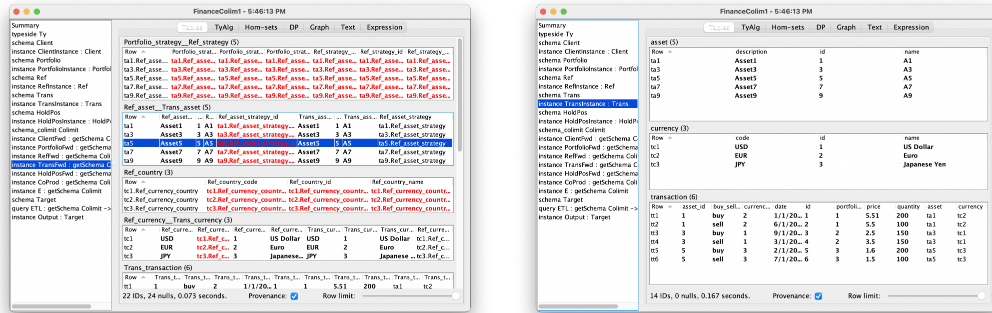

Figure 15: Lineage Tracing Example

Of course, the lineage of individual records leaves much implicit; for example the lineage described above does not directly mention the fact that the Trans table was linked and merged with the Ref table; that fact is implicitly accounted for by the "Ref-asset-strategy" component of the lineage. As such, the provenance check-box is not a substitute for a provenance API that allows the combined lineage of all records to be queried directly.

7 Appendix: Trickle Feed / Incremental Load

In many cases, after we construct our data warehouse we may wish to maintain it as new records are added to the source systems. To the extent possible, our technology supports trickle loading; however, it is important to note that trickle loading doesn't work with for example cyclic foreign keys; such data must be trickled in multiple rows (a "referentially closed group") at a time. The formal guarantee is that our technology can trickle in such groups; formally, we have equations such as $\Sigma_F(I + J) = \Sigma_F(I) + \Sigma_F(J)$.