

Urs: a timeseries engine

Semantics and computation for timeseries

1 Motivation

Forecasters commonly work with large tables whose timeseries structure and semantics are described in prose and encoded in dataset-specific scripts.

At The Forecasting Company, timeseries preparation grew organically alongside training pipelines, evaluation workflows, and customer-facing applications. Each developed its own scripts and SQL queries. Logic was duplicated, and behavior differed across the stack.

Foundation models and agentic systems create opportunities to work across a wider range of datasets and forecasting tasks. Supporting these workflows requires a systematic way to prepare timeseries, rather than a new script for each dataset and request.

We developed Urs to provide that foundation: an explicit semantic model of timeseries, from which operations such as aggregation, imputation, and covariate alignment can be derived.

We first illustrate these requirements through a retail forecasting example.

2 A Driving Example

Consider a retail chain with three datasets: sales transactions, daily store traffic, and hourly regional weather observations. The tables below contain illustrative records from March 2026. All timestamps are shown in UTC.

2.1 Sales transactions

Each row records a product sold at a store. Stores upload provisional transactions, which become visible in the central reporting system only after validation and reconciliation. Processing delays vary: a sale may appear several hours later, or the next day. Alongside the transaction time, the system records when the validated record became available as `available_at`.

Transaction time	Available at	Store	City	Country	SKU	Product family	Color	Units
2026-03-02 09:12	2026-03-02 11:40	S01	Paris	France	A01	Shirts	Blue	2
2026-03-02 09:18	2026-03-02 12:05	S01	Paris	France	A02	Shirts	Red	1
2026-03-02 10:05	2026-03-02 12:30	S02	Lyon	France	A01	Shirts	Blue	4
2026-03-02 11:32	2026-03-02 14:15	S03	Berlin	Germany	B01	Shoes	Blue	7
...	...	...	...	...	...	...	...	...

2.2 Store traffic

Entrance counters report the number of visitors to each store each day, regardless of whether they make a purchase.

Date	Store	City	Country	Visitors
2026-03-02	S01	Paris	France	842
2026-03-02	S02	Lyon	France	516
2026-03-02	S03	Berlin	Germany	731
2026-03-03	S01	Paris	France	795

2.3 Regional weather

A weather provider reports hourly observations by region. A separate geographical mapping associates each store's city with a weather region.

Timestamp	Region	Temperature (degrees C)
2026-03-02 00:00	Auvergne-Rhone-Alpes	4.3
2026-03-02 00:00	Berlin-Brandenburg	1.1
2026-03-02 00:00	Ile-de-France	6.2
2026-03-02 01:00	Auvergne-Rhone-Alpes	4.0
...	...	...

3 From Tables to Timeseries

Preparing these tables for forecasting requires more than matching columns. We need to identify the series they contain, the levels of detail they support, and the rules for combining their values. We introduce these concepts using the sales, traffic, and weather data above. Along the way, we build a description of the sales family in plain English. The cards collect what we have established, with new entries highlighted; together, these facts and choices form its schema.

3.1 Timeseries Family

Plotting the weather observations by region separates the dataset into three timeseries, each recording temperature over time.

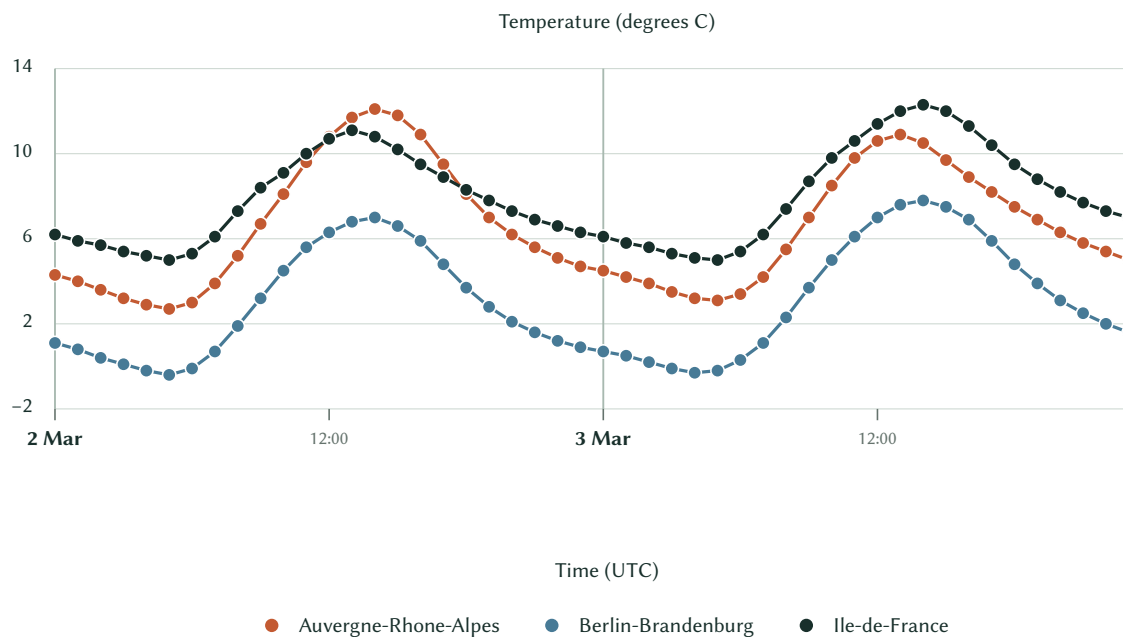


Figure 1: Regional weather observations from the example dataset. Each point is a CSV record; colors distinguish regions.

A single table can contain several related timeseries. In this example, each region has its own temperature timeseries; the **Region** column distinguishes them and is therefore an **identifier column**. Together, these series

form a **timeseries family**: a collection of related timeseries recording the same quantities and distinguished by identifier values.

A family may require several identifier columns: in the sales dataset, a store and a SKU together identify a timeseries.

3.2 The Structure Behind Identifiers: Hierarchies

In the sales dataset, knowing a store tells us its city and country. The reverse does not hold: a city can contain several stores. We call this relationship a **functional dependency**: **Store** determines **City**, and **City** determines **Country**. This is a property of the dataset's structure, not merely a pattern in the few rows displayed above.

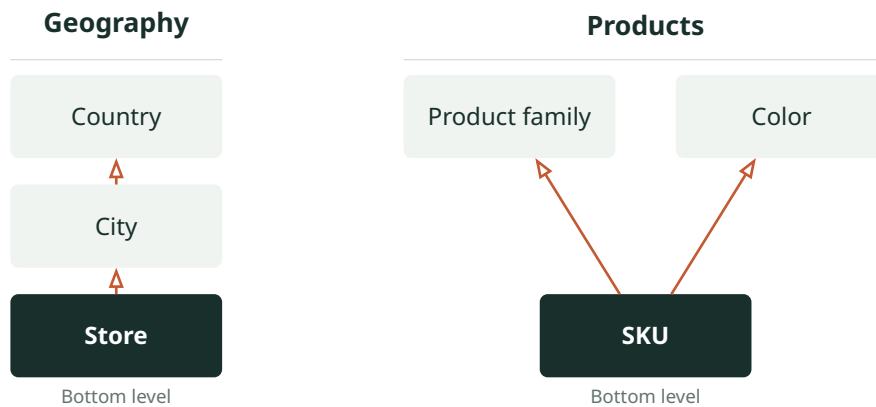


Figure 2: Each arrow points from an identifier to a value it uniquely determines. Product family and color form separate branches.

These dependencies form a **hierarchy**, with **Store**, **City**, and **Country** as its **hierarchy levels**. The most specific level, here **Store**, is the **bottom level**. Moving from stores to cities or countries gives us broader groupings of the same observations.

Products can be grouped in more than one way: by product family or by color. Neither grouping contains the other—shirts can have several colors, and blue products can include both shirts and shoes. A SKU determines both its product family and its color, so the hierarchy branches: **SKU** -> **Product family** and **SKU** -> **Color**. These offer different ways to group products, rather than successive levels of one chain.

The sales timeseries family has both geographical and product hierarchies. The traffic timeseries family shares the geographical hierarchy, but has no product hierarchy: visitors are counted by store, regardless of which products they buy.

Our timeseries family so far

+ **Hierarchies** Store → City → Country; SKU → Product Family or Color



3.3 Time and Its Granularities

Timeseries differ in what their observations refer to in time. A **punctual** observation refers to an instant, such as a temperature reading or a sales transaction. A **bucketed** observation refers to a time range: for instance, daily sales are observations that cover full days. We call this instant or range the observation's **target time**.

This distinction depends on what the observation represents, not on the quantity alone: a sales transaction is punctual, while a daily sales total refers to a range. Likewise, an instantaneous temperature reading and a daily average temperature have different temporal meanings.

Both structures can be regular or irregular. Hourly temperature readings occur at evenly spaced instants, whereas sales transactions occur at irregular times. Daily sales totals occupy regular calendar buckets.

For an example outside retail, consider electricity consumed during individual machine runs. Each observation covers one run, whose start time and duration can vary. The separate final track below illustrates these irregular ranges; it is not part of the retail dataset.

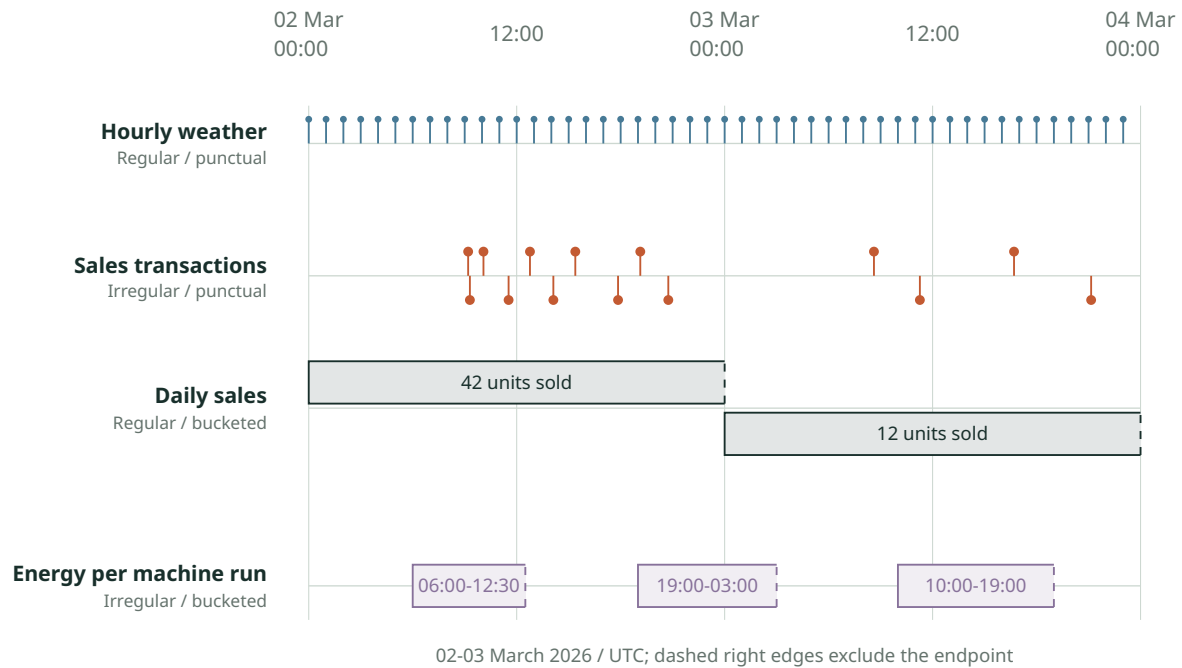


Figure 3: Two daily buckets, staggered vertically for readability, represent one series. Totals are computed from the sales CSV. Solid left and dashed right edges denote included starts and excluded ends: `[start, end)`.

3.4 Choosing a Granularity

The same sales dataset can be viewed at different levels of detail. We might want daily sales for each store and SKU, or weekly sales grouped by country and color. Each choice defines a **granularity**: one level from each hierarchy we want to retain, including the time hierarchy.

For example, we can request units sold at the following granularities:

- **Daily, store, SKU**: one daily total for each store–SKU pair.
- **Daily, city, product family**: daily totals combining stores within each city and SKUs within each product family.
- **Weekly, country, color**: weekly totals combining stores within each country and products of the same color, across product families.
- **Daily**: one daily total across all stores and products.

Leaving a hierarchy out means aggregating over it entirely. Keeping a coarser level instead preserves distinctions at that level: daily sales by city still form a family of city-specific timeseries, whereas daily sales with no geographical or product level form a single timeseries for the chain.

A granularity specifies levels, not particular identifier values. Daily sales by city describes a granularity; restricting those sales to Paris adds a separate filter. For units sold, the totals above are sums. Other observed quantities require their own rules for combining values.

Our timeseries family so far

Hierarchies Store → City → Country; SKU → Product Family or Color
+ Time Irregular instants; daily or weekly buckets after aggregation



3.5 Variates: Values, Availability, and Aggregation

A **variate** is a named quantity or attribute of a given type observed over time. Units sold, visitor count, and temperature are variates in our example. An **observation** records one value of one variate of a timeseries, together with its target time and known time.

An observation is not a whole table row. If a store's daily report contains both units sold and revenue, these are two separate observations grouped in the same row. They share a target day, but may become available at

different times. A variate's description specifies how its observations become known and how their values can be combined at different granularities.

3.5.1 When an Observation Becomes Known

An observation's target time tells us what instant or range it refers to. Its **known time** is the earliest instant at which the observation is available for use. Consider a daily sales report: stores close their accounts overnight, and validated totals are published at 06:00 the next morning. Under this illustrative reporting policy, the 42 units sold on 2 March refer to that whole day, but the total becomes known only at 06:00 on 3 March. The 12 units sold on 3 March become known at 06:00 on 4 March.

An **information cutoff datetime** is an instant defining what information is available. An observation is available at the cutoff if its known time is at or before it; observations known later are excluded or their values masked. This rule applies to individual observations, not necessarily to whole rows. At midnight on 4 March, both days have ended, but only the first daily sales observation has been published. Using the second in a forecast made at that cutoff would introduce information that was not available then. Target time alone is therefore insufficient to decide whether an observation can be used.

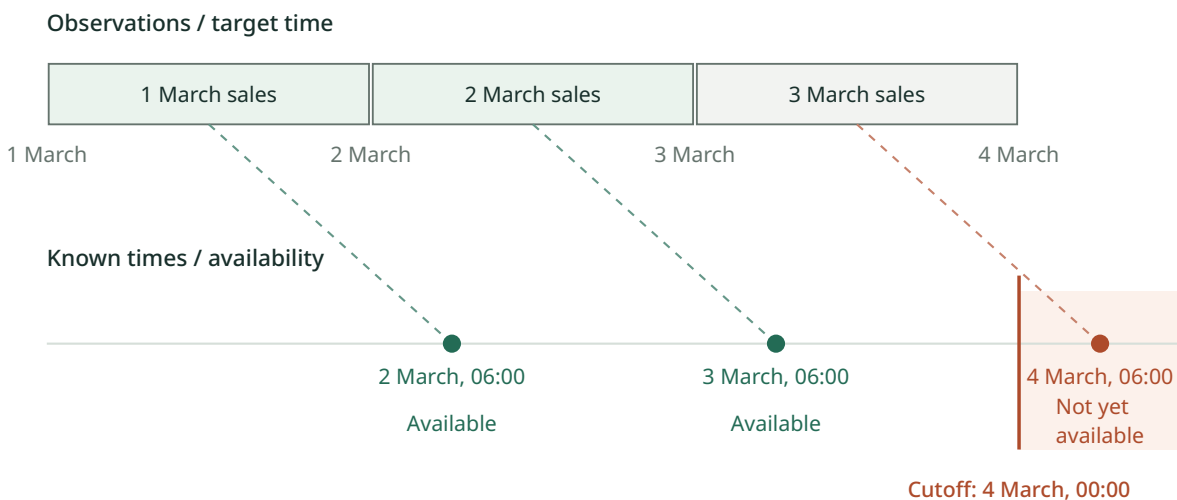


Figure 4: All three days have ended; only two observations are known at the cutoff.

Known time can lag behind target time, as with validated sales, or precede it, as with a weather forecast issued for a future day. A measured temperature and a forecast temperature may refer to the same hour while having different known times.

3.5.2 Different Hierarchies, Different Rules

Aggregation is a property of the variate and the hierarchy being traversed. For temperature, we might summarize each region's hourly readings as a daily tuple: (*minimum*, *average*, *maximum*). This changes both the temporal granularity and the value type, from one number to three. Sales, by contrast, use a sum to obtain daily units sold.

Aggregating over time and across places answers different questions. A daily temperature summary describes the weather shoppers experience; a national average probably carries no useful information. Consider weather as an input to forecasting store sales. The maps below show cities in France colored by their temperature on the same day, then progressively aggregated over geography. This information still makes sense at the department level. At the regional level, it's a little less clear: coastal, inland, and mountain conditions begin to blur together. Clearly, the weather signal aggregated across France is almost certainly useless.

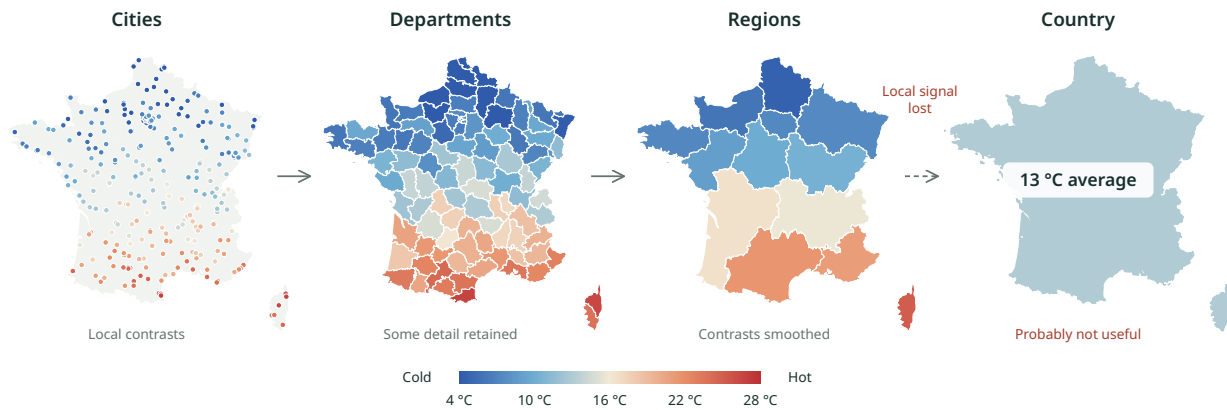


Figure 5: Local weather contrasts fade with geographical aggregation (illustrative temperatures).

Here the question is not technical: we know how to compute an average. The question is whether it makes sense to allow aggregation of weather data in our scenario. That's a choice we have to make.

The schema records that choice. For our weather family, we choose to support daily temperature summaries by region, but not to combine them across regions. Consequently, when the region hierarchy is removed, temperature is omitted from the result. This is a declared boundary of this schema, not an inherent limit on aggregating temperature.

This absence differs from a missing reading. A missing hourly temperature is a gap in an otherwise defined signal. Here, temperature at the coarser granularity has deliberately been left undefined: filling a missing value cannot supply a meaningful aggregation rule.

Some variates are computed from others. Average selling price, for example, is revenue divided by units sold. Its definition must specify the granularity of that calculation: dividing weekly revenue by weekly units sold generally differs from averaging daily selling prices. These are different quantities, not interchangeable ways to compute the same result.

Moving to a finer granularity also requires an explicit rule. Hourly temperatures might be estimated from readings taken every three hours, while allocating a daily sales total across hours requires an assumption about when sales occurred. Neither operation recovers the original observations: it produces estimates whose meaning depends on the chosen rule. In our model, **interpolation** changes granularity; **imputation** fills missing values at an already chosen granularity.

3.5.3 Availability of Derived Observations

Known time also follows the computation's dependencies. An aggregated observation cannot become known before its latest required input observation. The known time of an aggregated observation is thus the latest known time of its input. A completed day does not, by itself, make its sales total available if some transactions are still awaiting validation. Likewise, a daily weather summary depends on the availability of the readings required by its definition. A separate validation or publication step can delay availability further, as in the daily reporting policy above.

These rules describe the semantic contract. Representing known time and enforcing an information cutoff during execution are separate capabilities; the presence of availability metadata alone does not guarantee leakage-free preparation.

Our timeseries family so far

Hierarchies	Store → City → Country; SKU → Product Family or Color
Time	Irregular instants; daily or weekly buckets after aggregation
+ Aggregation	Sum over time, stores, and products
+ Known time	After transaction validation and publication



Date	Store	City	Product family	Units	Visitors	Temperature (min, mean, max)
2026-03-02	S01	Paris	Shirts	11	842	(5, 7.7, 11.1)
2026-03-02	S01	Paris	Shoes	3	842	(5, 7.7, 11.1)
2026-03-02	S02	Lyon	Shirts	11	516	(2.7, 6.7, 12.1)
2026-03-02	S02	Lyon	Shoes	1	516	(2.7, 6.7, 12.1)
2026-03-02	S03	Berlin	Shirts	9	731	(-0.4, 2.8, 7)
2026-03-02	S03	Berlin	Shoes	7	731	(-0.4, 2.8, 7)
2026-03-03	S01	Paris	Shirts	9	795	(5, 8.3, 12.3)
2026-03-03	S02	Lyon	Shirts	2	NULL	(3.1, 6.5, 10.9)
2026-03-03	S03	Berlin	Shoes	1	NULL	(-0.3, 3.3, 7.8)

Table 1: Daily sales joined with store traffic and regional weather, computed from the example CSVs. Temperatures are in degrees Celsius, rounded to one decimal place. No information cutoff is applied.

3.6 Bringing the Families Together

Suppose we want daily sales by store and product family, alongside the number of visitors and the day's weather. The three source datasets do not share the same granularity: sales are transactional and distinguish SKUs, traffic is daily by store, and weather is hourly by region.

We first sum sales within each day, store, and product family. Traffic already has the required time granularity, so we match it by date and store. For weather, we compute each region's daily (*minimum, average, maximum*) temperature, then use the city-to-region mapping to attach that summary to each store's sales. The resulting table retains the sales family's identifiers.

On 2 March, S01 has separate sales totals for shirts and shoes, but both rows carry the same 842 visitors and the same regional weather summary. These attached observations describe the store and region, respectively; repeating them beside product-family sales does not make them product-specific. Summing the visitor column across those two rows would count the same visitors twice.

The join also preserves sales without matching traffic. S02 and S03 have sales on 3 March but no traffic record for that day, so their visitor values are *NULL*, not zero. The sales observations remain present.

Joining families therefore involves both matching identifiers and preparing compatible granularities. Computing the regional weather summary before attachment also avoids weighting temperature by the number of sales rows. The rules governing an attached variate remain relevant after the join: attachment alone does not authorize summing visitor counts across products or averaging temperature across regions.

4 Urs in Practice

You have sales by store, but need sales by country. With Urs, you describe how the data is organized and how it should be aggregated, then ask for the timeseries you need. The same description can serve requests at different granularities, without a separate preparation script for each one.

The sales-family cards have already collected the meaning of such a schema in plain English. Urs records it in TOML or JSON: the source, identifier hierarchies, temporal structure, variates, and transformation rules. A query selects the desired granularity and optional filters; Urs translates the request into DuckDB SQL and executes it. The examples below follow a sales family from its schema to the prepared timeseries.

4.1 Describing a Family

For a compact example, we now use sales already summed by day and store, rather than the individual store-and-SKU transactions described above. The daily totals are stored in a Parquet file on S3 with five columns: *date*, *store*, *city*, *country*, and *units*. The S3 paths below are illustrative. We describe this family in *sales.toml*:

```

sales.toml

description = "Daily units sold by store"
has_gaps    = true

[source]
parquet = "s3://example-retail/sales.parquet"

[columns]
store = { kind = "identifier", type = "string" }
city  = { kind = "identifier", type = "string" }
country = { kind = "identifier", type = "string" }
units = { kind = "variate", type = "int64", aggregate = "sum" }

[columns.date]
kind      = "time"
type      = "date32"
frequency = "daily"
default_partitions = false

[hierarchies]
geo = ["store", "city", "country"]

```

The schema declares daily time buckets, the store–city–country hierarchy, and summation for units sold. Missing values are left unfilled.

4.2 Discovering the Structure

A schema need not be written entirely by hand. Urs can inspect a Parquet source to propose identifier hierarchies from functional dependencies, identify a time column, and infer its cadence and alignment. For example, repeated store, city, and country values provide evidence for the geographical hierarchy introduced above:

```
urs infer s3://example-retail/sales.parquet --output sales.toml
```

The result is a draft to inspect and refine, not a substitute for domain knowledge. Dependencies observed in the data may be accidental, particularly in small samples. Aggregation and imputation choices remain explicit: the data alone does not tell us whether a quantity should be summed, averaged, or filled with zero.

4.3 Composing Two Families

Daily store traffic is stored in another Parquet file on S3, described by `traffic.toml` with the same hierarchy and a `visitors` variate. We attach it to sales by matching date and store. The composed schema, `sales-with-traffic.toml`, references both families:

```

sales-with-traffic.toml

ref = "sales.toml"

[[covariates]]
ref = "traffic.toml"
on  = ["date", "store"]

```

The base family retains its identifiers and sales observations. The attachment adds visitors, leaving it missing where no traffic record matches.

4.4 Requesting a Materialization

To obtain daily sales by country, retain the daily level and the country level. The schema supplies the aggregation rule:

```
urs materialize sales.toml 'date=*,country=*' --layout rank-columns
```

To obtain daily sales and traffic by store, query the composed family instead:

```
urs materialize sales-with-traffic.toml 'date=*,store=*' --layout rank-columns
```

* retains all values at a level. The layout option displays identifiers as separate columns. The expected joined values are:

Date	Store	Units	Visitors
2026-03-02	S01	14	842
2026-03-03	S01	9	795
2026-03-02	S02	12	516
2026-03-03	S02	2	NULL
2026-03-02	S03	16	731
2026-03-03	S03	1	NULL

Table 2: Expected daily sales and traffic by store. This is an illustrative result, not captured CLI output. **NULL** denotes missing traffic.

The same family descriptions support different granularities and joins without rewriting their preparation logic. These examples cover values and aggregation; they do not apply an information cutoff.

5 Under the Hood

How does Urs turn a schema and a request into SQL? This section follows a request through the compiler, from loading the schema to executing the generated query.

5.1 From Request to Result

Consider the daily-sales-by-country request from the preceding section. Its inputs are a family schema, the selector `date=*,country=*`, and an output layout. Figure 6 distinguishes the representations produced along the way from the processing steps that transform them.

1. **Load and validate.** The `schema-format` crate decodes the TOML or JSON document. The `schema-desugar` crate resolves references and expands shorthand into the model defined by the `core` crate. Validation checks types, hierarchy structure, and dependencies, producing a well-formed family schema.
2. **Build the plan.** Given that schema, the selector, and the output layout, the `materialize` crate finds transformation paths and schedules their dependencies. It lowers those computations into a relational **intermediate representation** (IR): a graph of operators such as scans, aggregations, joins, and projections. The simplified chain shown here scans sales, aggregates by day and country, and selects the output columns.
3. **Optimize the IR.** Optimization passes simplify compatible calendar expressions, prune unused columns, remove neutral padding where safe, ... The compiler checks the plan's structure before and after these rewrites. This is a single sequence of passes, not a loop to convergence.
4. **Emit SQL.** The compiler renders the reachable operators as SQL, using common table expressions (CTEs) to connect intermediate computations. DuckDB then chooses the physical execution plan and runs the query. Results can be consumed as Arrow batches or written to files through `COPY`.

A top-level library connects compilation and execution, while a command-line interface exposes them as commands. Compilation and execution remain decoupled, so SQL can be generated independently of materialization.

Schema discovery sits upstream of this path. The `discovery` crate inspects sources to propose identifier hierarchies and temporal structure; its output can help construct the schema subsequently loaded by a preparation request.

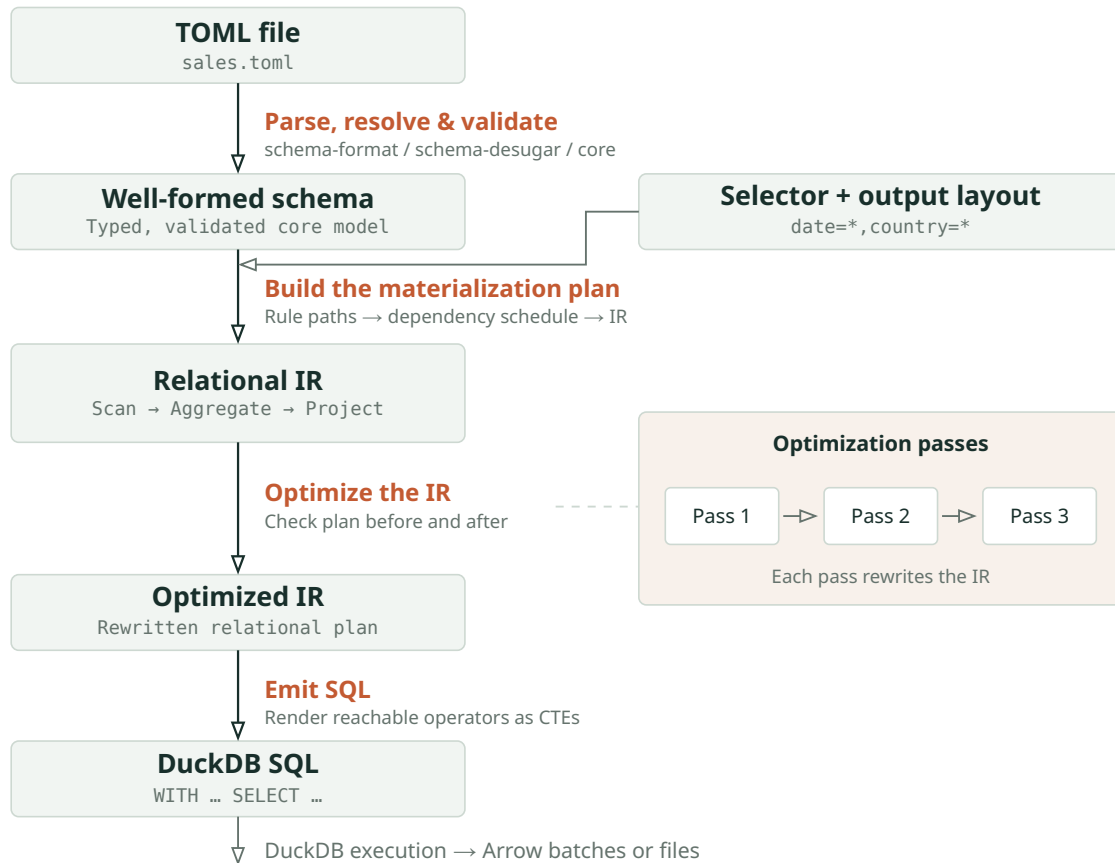


Figure 6: Compilation transforms a validated schema and a request into SQL. The operator chain is illustrative.

5.2 Planning the Variates

Building the plan means deciding where to compute each variate and how to bring its inputs together. The requested output granularity is not necessarily the granularity at which every computation must happen. Consider monthly electricity cost by city. One family records store consumption in kWh at 15-minute intervals; a separate family records hourly tariffs in €/kWh by country. Assume that each hourly tariff applies to all cities in its country, with complete tariff coverage.

Figure 7 shows selected temporal and spatial granularities: 15-minute, hourly, or monthly, by store, city, or country. Each dot denotes a granularity, not an observation. Consumption starts at 15-minute × store, while the tariff belongs to a separate family at hour × country. The requested output is at month × city.

First, consumption is summed within each hour and across stores in each city, producing **hour × city** totals. The schedule attaches the matching hourly country tariff, broadcasting it to the cities in that country, and computes **electricity cost = consumption × tariff** at that granularity. These hourly costs are then summed to **month × city**.

The order matters when tariffs change during the month. Summing consumption to the month first would lose which kWh were consumed at which tariff; multiplying monthly consumption by an unweighted average tariff would not in general recover the cost. The schema therefore declares the cost expression at hour–city, followed by summation to month–city. The planner follows those definitions rather than selecting a pricing policy itself. If preparation requires completion or imputation, those steps follow the input family’s declared rules before attachment. An unmatched tariff is not automatically zero: treating it as such would understate cost. The complete-coverage assumption here keeps that separate policy out of the illustrated schedule.

Internally, each variate’s rules define possible transitions between granularities. Path selection uses depth-first search with backtracking, trying constrained rules before unconstrained fallbacks rather than comparing execution costs. Derived variates recursively require their inputs at the granularity where their expressions are evaluated. The scheduler combines these paths and dependencies into stages, placing producers before consumers. This dependency schedule is then lowered into the relational IR.

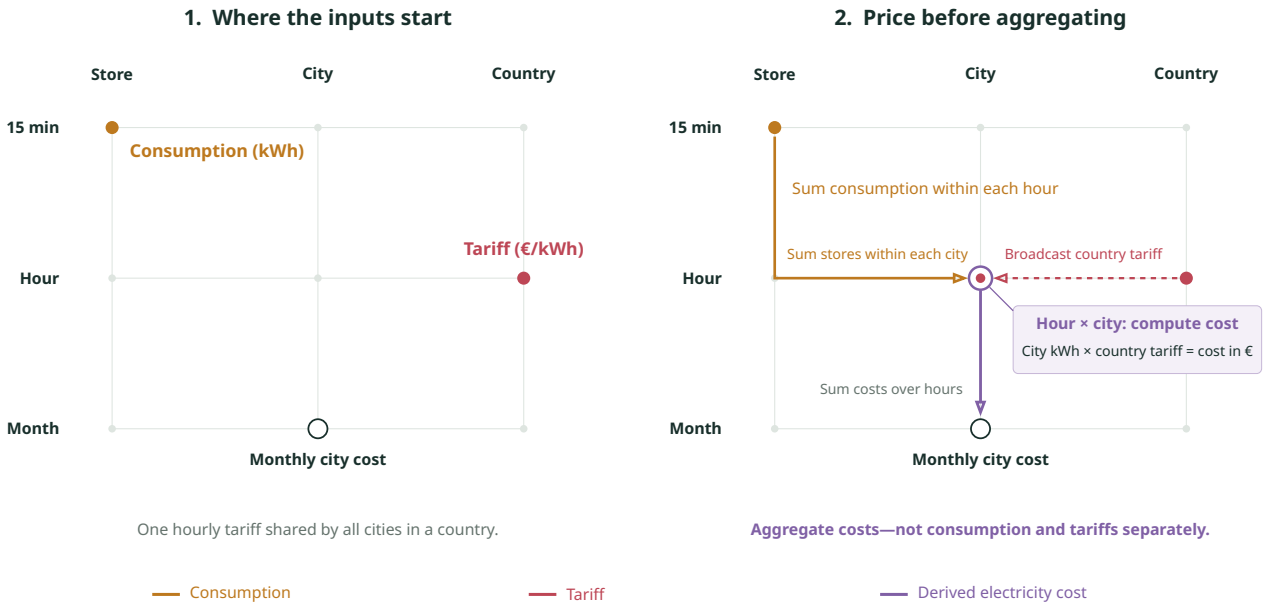


Figure 7: Consumption aggregates to hour × city; the country tariff broadcasts to that level. Compute costs, then sum to month × city.

5.3 The Intermediate Representation

Before SQL generation, a request is represented as a directed acyclic graph of relational operators. In Rust, a **Plan** stores a vector of **Op** enum values; operators refer to their inputs by node index. An aggregation records its input node, grouping keys, and aggregate expressions, rather than a fragment of SQL text.

Figure 8 shows a simplified plan for daily sales by country, with each operator’s output **frame type**: its column names and row domain. Here, **Observed** distinguishes frames that do not carry completion provenance; a frame completed on a time grid instead records **Complete(spine_id)**. The IR is typechecked before and after optimization.

The aggregation reads **date**, **country**, and **units**, but not **store**. Column pruning therefore removes **store** from the scan. Its intermediate frame type changes, while the aggregate retains the same output columns and row domain. The rewrite preserves the result because the removed column is unused; matching final types alone would not establish that equivalence.

The IR combines familiar relational operators with explicit timeseries preparation steps:

- **Scan**, **Filter**, **Project**, and **Aggregate** read and reshape data.
- **Derive** computes expressions; **Window** applies time-ordered filling within each series.
- **Spine** constructs a temporal grid; **Complete** aligns observed data to it, introducing missing rows.

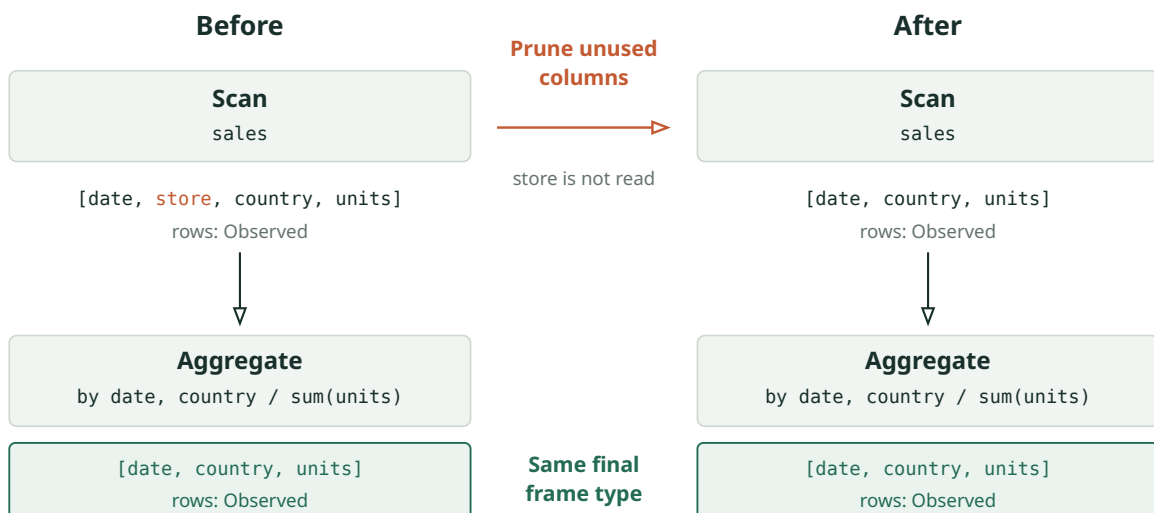


Figure 8: Unused-column pruning changes the scan’s frame type, but preserves the final frame type. Simplified plans; annotations show columns and row domain.

- **Join** connects prepared families; **Merge** combines intermediate results completed on a shared grid.

This representation lets optimization passes rewrite computations before any SQL is emitted. Node indices remain stable during rewriting; bypassed nodes can become unreachable, and SQL generation visits only those needed for the result.

6 Conclusion

We presented our timeseries engine *Urs*. It gives a dataset an explicit timeseries structure: how individual series relate to one another, how their identifiers form hierarchies, what their observations refer to in time, and how those observations can be combined and used. This description forms the family's schema.

Once this schema is declared, users request the timeseries they need. *Urs* determines how to prepare them through completion, imputation, aggregation, and interpolation, without a separate script for each request.

Future work will strengthen support for punctual and irregular timeseries, alongside the regular series most forecasting models expect. We also plan to explore richer observation values. In booking data, for example, each service date carries not just a total, but a history of incoming reservations—a timeseries in its own right. Preserving this history could help forecast attendance for that date. Longer-term, we would like to extend the same approach to observations containing text or images.

