**Weather Data Management System – Database Design Document**

**Mission and Business Problems**

Mission: Provide a reliable and normalized database system to collect, process, and report weather data. The system captures real-time weather observations from stations and sensors, stores forecast runs and forecasted values, manages severe-weather alerts, and supports reporting for analysts, operators, and decision-makers.

Business Problems Being Addressed:  
• Data silos: Observations, forecasts, and alerts are often stored separately, causing difficulty in reporting.  
• Lack of traceability: Observed values lack provenance (station/sensor), reducing trust and quality checks.  
• Inefficient querying: High-volume time-series data leads to performance issues without proper design.  
• Operational gaps: Maintenance events and alert history are not tied to observations and forecasts.  
• Role-based access: Different user roles (admins, analysts, operators) require structured access.

**Entities, Why They Are Included and Relationships Between Entities**

1) Location

Why it exists: Anchors all weather activity to a geographic place (city/county/campus/site). Needed for coverage analysis, regional alerts, and roll-up reporting.  
Key relationships:

* One Location has many WeatherStations
* One Location has many Alerts   
  Why not merge with Station: Multiple stations can serve the same area; alerts are often region-wide, not station-specific.  
  Design notes: (name, country, lat, lon, elevation). Index by (country, name) and spatial coords for radius queries.

2) WeatherStation

Why it exists: Represents a physical station that hosts sensors and produces observations and forecasts. Central operational unit.  
Key relationships:

* One Location has many WeatherStations
* WeatherStation (One to Many) Sensor, Observation, Forecast, StationMaintenance  
  Why not fold into Sensor: Station lifecycle, availability, and maintenance are distinct from individual sensors.  
  Design notes: (station\_id, name, status, installed\_date).

3) Forecast

Why it exists: Captures a single forecast run (issued\_at, valid window, model/station scope). Separating the “run” from its time-step values keeps the fact table thin and queryable.  
Key relationships:

* One WeatherStation has many Forecasts
* Forecast (One to Many) ForecastValue  
  Why not store values here: Forecasts produce many time steps and variables; normalizing avoids wide rows and update anomalies.  
  Design notes: (forecast\_id, issued\_date, valid\_from, valid\_to).

4) Alert

Why it exists: Stores actionable weather warnings (time-bounded, severity, message) bound to a geographic Location.  
Key relationships:

* AlertType (One to Many) Alert
* Location (One to Many) Alert  
  Why not attach to Station: Alerts are typically regional and may cover multiple stations; binding to Location supports that.  
  Design notes: (alert\_id, start\_time, end\_time, message). Constraints: starts\_at < ends\_at.

5) User

Why it exists: Access control and accountability (who views, acknowledges, or manages alerts/stations).  
Key relationships:

* User … optionally tied to roles/organizations (out of scope if keeping minimal)  
  Why needed now: Even in MVP, you’ll need at least basic authentication/authorization and audit trails for changes.  
  Design notes: (user\_id, name, email UNIQUE). Extend later with acknowledgements or subscriptions.

6) Role

Why it exists: Enables simple, auditable role-based access (admin/analyst/operator) so only the right people can create alerts, approve QC overrides, or manage stations.

Key relationships:

One Role has many Users

Each User belongs to exactly one Role

Why not merge with User: Hard-coding permissions on each user makes policy changes painful and inconsistent. A central Role lets you update once and apply everywhere.

Design notes: (role\_id, name UNIQUE, permission)

7) Sensor (with subtypes & measurement units)

Why it exists: Models the physical devices producing readings (e.g., Thermometer, Hygrometer, Anemometer) with subtype-specific attributes and native units.  
Key relationships:

* WeatherStation (One to Many) Sensor
* Sensor (Optional one to Many) Observation (optional sensor\_id allows station-level derived obs)  
  Why subtypes: Different calibration rules, ranges, and units by sensor type (°C, %RH, m/s, hPa). Subtyping prevents null-heavy generic fields and supports validation.  
  Design notes: (sensor\_id, model, calibrated\_date). Consider a reference table for allowed units per subtype.

8) SensorType

Why it exists: Canonical catalog of hardware types (Thermometer, Hygrometer, Anemometer…) with unit, safe ranges, precision, and calibration cadence for validation and maintenance planning.

Key relationships:

One SensorType has many Sensors

Each Sensor belongs to exactly one SensorType

Why not merge with Sensor: Unit/range/precision would be duplicated across thousands of sensors and drift over time. Centralizing in SensorType keeps QC rules and maintenance logic consistent.

Design notes: (sensor\_type\_id, name UNIQUE, unit, min\_range, max\_range). CHECK min\_range < max\_range.

9) MeasurementType

Why it exists: Standardizes what is being measured (TEMP, RH, WIND, PRESS), along with default units and QC bounds. This lets you compare data across stations/sensors and align observations with forecasts without guessing from free text.

Key relationships:

One MeasurementType has many Observations

One MeasurementType has many ForecastValues

Why not merge with SensorType: The same variable (e.g., TEMP) can come from different hardware types; forecasts also use variables independent of device. Keeping it separate avoids duplication and keeps analytics clean.

Design notes: (measurement\_type\_id, code UNIQUE, description, default\_unit).

10) ForecastModel

Why it exists: Identifies the forecasting engine/version/source that produced a run. Critical for reproducibility, performance comparisons (MAE/RMSE), and governance when configs change.

Key relationships:

One ForecastModel has many Forecasts

Each Forecast belongs to exactly one ForecastModel

Why not merge with Forecast: You’d repeat the same model metadata on every run and make “accuracy by model/version” slow and error-prone. Separating keeps runs lightweight and analysis simple.

Design notes: (model\_id, name, version, forecast\_duration). Unique (name, version).

11) AlertType (with subtypes like Red/Orange)

Why it exists: Standardizes policy/severity logic (Red/Orange/Yellow), enables consistent UI coloring, escalation, and filtering.  
Key relationships:

* AlertType (One to Many) Alert  
  Why separate: Severity semantics (thresholds, SLA, iconography) are stable and reused; storing on alerts directly risks duplication and inconsistency.  
  Design notes: (alert\_type\_id, code, severity, description). Add constraints so only defined codes (e.g., RED/ORANGE/YELLOW) are allowed.

12) StationMaintenance

Why it exists: Tracks calibrations, repairs, and outages that affect data quality and availability. Critical for audit and QC.  
Key relationships:

* WeatherStation (One to Many) StationMaintenance  
  Why separate from Observation: Maintenance events are sparse, descriptive, and station-level; mixing with observations would pollute fact tables.  
  Design notes: (maintenance\_id, performed\_on, notes, technician).

13) ForecastValue

Why it exists: The time-series outputs of a forecast run per variable (e.g., hourly temp next 48h). Core fact table for forecast analytics.  
Key relationships:

* Forecast (One to Many) ForecastValue
* Optionally link to Measurement/variable code for consistency across models  
  Why separate from Observation: Observed vs predicted are different processes with different lineage; separation enables error metrics (MAE/RMSE) cleanly.  
  Design notes: (forecast\_value\_id, time, value, unit). Constraints: for\_time within Forecast.valid window.

14) Observation

Why it exists: The canonical time-series of measured weather data (temp, humidity, wind, etc.). Foundation for dashboards and verification.  
Key relationships:

* WeatherStation (One to Many) Observation
* Sensor (Optional one to Many) Observation (nullable sensor\_id for station-level aggregates)  
  Why not store on Sensor: Observations are many-to-one over time; putting them on Sensor would bloat rows and complicate queries.  
  Design notes: (observation\_id, observed\_time, value, Quality\_Flag\_Control).

**Key Database Design Decisions**

Separate MeasurementType from SensorType to distinguish 'what is measured' vs 'what device measures it'.

Optional sensor\_id on Observation allows derived or aggregated values when no specific sensor is referenced.

Separate Forecast and ForecastValue so that forecasts scale with multiple variables and long horizons.

Attach Alerts to Location instead of Station to support region-wide warnings.

Use simple Role/User schema for initial role-based access control.

Keep StationMaintenance separate from observations for cleaner audit trails.

**Feasible Future Enhancements**

* **User Subscriptions:** Allow users to subscribe to specific locations or alert types (schema-level only, no full SMS/email build).
* **Daily Summaries:** Add a table for daily/min/max/avg observations per location to support faster reporting.
* **Forecast Evaluation:** Create a table to store forecast accuracy metrics (e.g., MAE, RMSE) by comparing forecasts vs. actual observations.
* **Indexing & Partitioning:** Document key indexes and partition Observations/ForecastValues by date for scalability.

**Conclusion**

The proposed ERD and database design solve core problems of weather data management by ensuring normalization, traceability, and performance. The chosen entities represent key operational, analytical, and security needs. This design is scalable, auditable, and supports both real-time monitoring and historical analysis.