# HTNQL

Hierarchical Task Network Query Language

*A Deep-Dive Technical Documentation*

Declarative Reporting over Relational Databases
with Automatic Join Inference via HTN Planning

Version 2.0
December 2025

## Abstract

This document provides a comprehensive technical reference for HTNQL (Hierarchical Task Network Query Language), a Python library that enables declarative reporting over relational databases. HTNQL distinguishes itself from traditional query builders by employing techniques from artificial intelligence planning—specifically, Hierarchical Task Network (HTN) planning—to automatically infer table joins from database schema metadata.

The document is structured as a dissertation-style deep-dive, beginning with theoretical foundations of HTN planning, proceeding through the architectural design and implementation details of the HTNQL library, and concluding with practical examples using the provided demonstration databases. The intended audience includes software engineers seeking to understand or extend the library, researchers interested in the application of AI planning techniques to database systems, and practitioners evaluating HTNQL for production use.

## Table of Contents

# PART I

Theoretical Foundations

# 1. Introduction to Automated Planning

Automated planning is a branch of artificial intelligence concerned with the realization of strategies or action sequences, typically for execution by intelligent agents, autonomous robots, or unmanned vehicles. The planning problem can be formally defined as follows: given a description of the initial state of the world, a description of the desired goal state, and a set of possible actions that can transform states, find a sequence of actions that transforms the initial state into a goal state.

## 1.1 Classical Planning

Classical planning operates under several simplifying assumptions that make the problem tractable while still capturing many real-world scenarios:

- Finite State Space: The world can be described by a finite set of state variables.
- Deterministic Actions: Each action has a single, predictable outcome.
- Full Observability: The agent has complete knowledge of the current state.
- Static Environment: The world changes only as a result of the agent's actions.
- Discrete Time: Actions are instantaneous and occur in discrete steps.

The classical planning problem is represented using state-space search, where nodes represent world states and edges represent actions. The goal is to find a path from the initial state to any state satisfying the goal condition.

## 1.2 The STRIPS Formalism

STRIPS (Stanford Research Institute Problem Solver) introduced a foundational representation for planning problems that remains influential today. In STRIPS, each action is described by:

- Preconditions: Conditions that must be true for the action to be applicable.
- Add Effects: Facts that become true after the action is executed.
- Delete Effects: Facts that become false after the action is executed.

While powerful for many domains, STRIPS and its descendants suffer from limitations when dealing with hierarchical problems—tasks that naturally decompose into subtasks. This limitation motivated the development of Hierarchical Task Network planning.

## 1.3 Limitations of Flat Planning Approaches

Traditional planning approaches treat all actions as atomic operations at the same level of abstraction. This "flat" approach encounters several difficulties:

1. Scalability: The state space grows exponentially with the number of state variables, making exhaustive search impractical for complex domains.
2. Knowledge Representation: Domain expertise about how to solve problems—procedural knowledge—cannot be directly encoded.
3. Plan Quality: Without hierarchical structure, planners may produce valid but suboptimal or unnatural plans.
4. Reusability: Solutions to subproblems cannot be easily composed or reused across different planning scenarios.

These limitations led researchers to develop planning formalisms that could capture hierarchical structure explicitly.

# 2. Hierarchical Task Network Planning

Hierarchical Task Network (HTN) planning represents a paradigm shift from goal-directed planning to task-directed planning. Rather than specifying a goal state and asking "how do I get there?", HTN planning starts with a high-level task and asks "how do I accomplish this?" The answer involves decomposing tasks into subtasks until only primitive, directly executable actions remain.

## 2.1 Fundamental Concepts

### 2.1.1 Tasks and Task Networks

In HTN planning, the fundamental unit is the **task**, which represents something that needs to be accomplished. Tasks are classified into two categories:

- Primitive Tasks: Actions that can be executed directly, analogous to operators in classical planning. Each primitive task has preconditions and effects.
- Compound Tasks: Abstract tasks that cannot be executed directly but must be decomposed into simpler tasks.

A **task network** is a set of tasks along with constraints on their ordering and variable bindings. The initial task network represents the problem to be solved, while a solution is a task network containing only primitive tasks that can be executed in sequence.

### 2.1.2 Methods

The key mechanism for decomposing compound tasks is the **method**. A method specifies one way to accomplish a compound task by replacing it with a network of subtasks. Each method consists of:

- Task: The compound task that this method can decompose.
- Preconditions: Conditions that must hold for this method to be applicable.
- Subtasks: The task network that replaces the compound task.

Multiple methods may apply to the same compound task, representing alternative ways to accomplish it. The planner explores these alternatives during search.

### 2.1.3 The HTN Planning Algorithm

The basic HTN planning algorithm proceeds as follows:

```
function HTN-Plan(state, task-network, methods, operators):
    if task-network is empty:
        return empty-plan

    select a task T from task-network

    if T is primitive:
        if preconditions of T are satisfied in state:
            apply T to state, yielding new-state
            plan ← HTN-Plan(new-state, task-network - T, methods, operators)
            if plan ≠ failure:
                return [T | plan]
        return failure

    else:  # T is compound
```

```
        for each method M applicable to T in state:
            new-network ← replace T with subtasks of M in task-network
            plan ← HTN-Plan(state, new-network, methods, operators)
            if plan ≠ failure:
                return plan
        return failure
```

## 2.2 Advantages of HTN Planning

HTN planning offers several significant advantages over classical planning approaches:

### 2.2.1 Encoding Domain Knowledge

Methods encode procedural knowledge—the "how-to" knowledge that experts use to solve problems. This knowledge guides the search process, dramatically reducing the search space by eliminating implausible decompositions.

### 2.2.2 Natural Abstraction Hierarchies

Many real-world problems have natural hierarchical structure. HTN planning allows this structure to be represented directly, producing plans that are more understandable and maintainable.

### 2.2.3 Computational Efficiency

By encoding domain-specific knowledge in methods, HTN planners can avoid exploring large portions of the search space. This makes HTN planning practical for problems that would be intractable for classical planners.

### 2.2.4 Flexibility in Plan Representation

The hierarchical structure of HTN plans provides flexibility in execution. Plans can be refined, adapted, or repaired at different levels of abstraction.

## 2.3 HTN Planning in HTNQL

HTNQL applies HTN planning to the domain of SQL query generation. In this context:

- The initial task is 'AnswerReport'—generate and execute a SQL query to satisfy a report specification.
- Compound tasks include 'ChooseExecutionMode', 'PlanExecution', and 'PlanAutoSql'.
- Primitive tasks include 'ValidateSpecStructurally', 'InferTablesFromSpec', 'FindJoinForest', 'BuildSqlFromPlan', and 'ExecutePlannedSql'.
- Methods encode different strategies for accomplishing tasks, such as strict FK-based joins versus heuristic joins.

This application of HTN planning to database queries is novel in that it treats SQL generation as a planning problem where the "world state" includes schema metadata and the report specification, and "actions" transform partial query plans into complete, executable SQL.

# 3. Formal Model of HTNQL Planning

This chapter presents a formal model of the HTN planning approach used in HTNQL, establishing the mathematical foundations for understanding the system's behavior and correctness.

## 3.1 Definitions

### 3.1.1 Planning State

A **planning state** $\sigma$ is a tuple:

$\sigma = (E, G, S, M, T, J, Q, R)$

where:

- $E$ is a database engine (connection)
- $G$ is a schema graph (tables and foreign key relationships)
- $S$ is a report specification (metrics, groupings, filters)
- $M$ is the execution mode $\in$ {raw_sql, base_sql, auto, $\perp$}
- $T$ is a set of inferred tables
- $J$ is a join forest (set of FK edges connecting tables)
- $Q$ is the generated SQL (AST or text)
- $R$ is the result set

### 3.1.2 Tasks

A **task** $\tau$ is a pair (name, args) where name is a task identifier and args is a possibly empty tuple of arguments.

### 3.1.3 Methods

A **method** $\mu$ is a tuple ($\tau$, cond, decomp, name) where:

- $\tau$ is the task name this method applies to
- cond: $\Sigma \times \tau \rightarrow$ {true, false} is a condition function
- decomp: $\Sigma \times \tau \rightarrow [\tau_1, \tau_2, ..., \tau_n]$ produces a sequence of subtasks
- name is a human-readable identifier

### 3.1.4 Primitive Operations

A **primitive operation** $\pi$ is a pair ($\tau$, apply) where:

- $\tau$ is the task name
- apply: $\Sigma \times \tau \rightarrow \Sigma$ transforms the planning state

## 3.2 Planning Algorithm

The HTNQL planner implements depth-first HTN planning with backtracking. Given an initial state $\sigma_0$ and a top-level task $\tau_0$, the planner computes:

$\mathrm{Plan}(\sigma_0, \tau_0) \rightarrow \sigma\_\mathrm{final}$

The algorithm maintains a stack of pending tasks and processes them as follows:

1. If $\tau$ is primitive: look up $\pi = (\tau, \mathrm{apply})$ and compute $\sigma' = \mathrm{apply}(\sigma, \tau)$
2. If $\tau$ is compound: find an applicable method $\mu$ where $\mathrm{cond}(\sigma, \tau) = \mathrm{true}$, then recursively plan each subtask from $\mathrm{decomp}(\sigma, \tau)$
3. Backtracking occurs when no method is applicable or a primitive operation fails

## 3.3 Correctness Properties

### 3.3.1 Soundness

The planner is **sound** if every plan it produces, when executed, yields a valid SQL query that correctly implements the report specification. Formally:

$\forall \sigma_0, \tau_0$: if $\mathrm{Plan}(\sigma_0, \tau_0) = \sigma\_\mathrm{final}$ and $\sigma\_\mathrm{final.R} \neq \perp$, then $\sigma\_\mathrm{final.R} = \mathrm{Execute}(\sigma\_\mathrm{final.Q})$ and $\sigma\_\mathrm{final.Q}$ $_0$ .S□

where ⊨ denotes that the query correctly implements the specification.

### 3.3.2 Completeness

The planner is **complete** relative to a method library if, for any satisfiable report specification, the planner finds a valid plan. HTNQL's completeness depends on:

- The schema graph containing all necessary FK relationships
- The report specification using properly qualified table.column references
- The method library covering all necessary decomposition patterns

# PART II

Architecture and Implementation

# 4. System Architecture

HTNQL is designed as a layered architecture with clear separation of concerns. This chapter provides a detailed examination of each architectural layer and its responsibilities.

## 4.1 Architectural Overview

The system consists of four primary layers:

1. Specification Layer: Defines the declarative report interface (ReportSpec, MetricSpec, FilterSpec)
2. Schema Layer: Represents database structure as a graph (SchemaGraph, FKEdge)
3. Planning Layer: HTN planner with methods and primitives (HTNPlanner, Method, PrimitiveOp)
4. Execution Layer: SQL generation and database interaction (QueryEngine)

Data flows through these layers as follows:

```
ReportSpec ──→ QueryEngine ──→ HTNPlanner ──→ PlanningState
                    │                              │
                    │           SchemaGraph ──────┤
                    │                              │
                    ←─────── SQL + Results ───────┘
```

## 4.2 Module Dependency Graph

The HTNQL library is organized into the following modules with their dependencies:

| Module | Primary Responsibility | Dependencies |
|---|---|---|
| report_spec.py | Data classes for report specifications | None (dataclasses only) |
| schema_graph.py | Graph representation of database schema | sqlalchemy.MetaData |
| htn_core.py | Core HTN planner implementation | None (pure Python) |
| planning_state.py | Mutable state for planning | schema_graph, report_spec, htn_core |
| planning_domain_basic.py | Core planning methods and primitives | htn_core, planning_state |
| planning_primitives.py | Low-level SQL building operations | sqlalchemy, planning_state |
| agent_dsl.py | DSL for defining planning agents | htn_core, planning_state |
| builtin_agents.py | Pre-defined agent configurations | None (dict config only) |
| shape_suggestion.py | Query shape recommendation | schema_graph |
| query_engine.py | Main entry point, orchestration | All of the above |

## 4.3 Design Principles

HTNQL's architecture follows several key design principles:

### 4.3.1 Declarative Specification

Users declare *what* they want (metrics, groupings, filters) rather than *how* to get it (joins, SQL syntax). This separation allows the system to choose optimal execution strategies.

### 4.3.2 Schema-Driven Intelligence

The system derives join paths from foreign key metadata rather than requiring explicit join specifications. This reduces user burden and ensures consistency with the actual database structure.

### 4.3.3 Extensibility via Configuration

Planning behavior is controlled through agent configurations (dictionaries) rather than code changes. New planning strategies can be added by defining new agents in the configuration.

### 4.3.4 Graceful Degradation

Three execution modes (raw_sql, base_sql, auto) provide escape hatches when automatic planning is insufficient. Users can always fall back to explicit SQL when needed.

# 5. Core Components

This chapter provides detailed documentation of each core component in the HTNQL library.

## 5.1 Report Specification (report_spec.py)

The report specification module defines three dataclasses that comprise the declarative interface for reports.

### 5.1.1 MetricSpec

Represents a single metric (aggregate expression) in the report's SELECT list.

```
@dataclass
class MetricSpec:
    expr: str   # SQL expression, e.g., 'SUM(order_item.line_total)'
    alias: str  # Output column name, e.g., 'total_revenue'
```

The `expr` field contains a valid SQL expression that can reference table-qualified columns. The expression is passed through to the generated SQL with minimal transformation.

### 5.1.2 FilterSpec

Represents a WHERE clause predicate.

```
@dataclass
class FilterSpec:
    column: str  # 'table.column' or just 'column' in base_sql mode
    op: str      # '=', '!=', '<', '>', '<=', '>=', 'IN', 'LIKE', etc.
    value: Any   # Literal value or list (for IN)
```

The supported operators and their semantics:

| Operator | SQL Equivalent | Value Type |
|---|---|---|
| = | column = value | Scalar |
| != | column != value | Scalar |
| < | column < value | Scalar |
| > | column > value | Scalar |
| <= | column <= value | Scalar |
| >= | column >= value | Scalar |
| IN | column IN (v1, v2, ...) | List |
| LIKE | column LIKE pattern | String with wildcards |

### 5.1.3 ReportSpec

The main specification class that combines all elements of a report.

```
@dataclass
class ReportSpec:
    name: str                       # Logical name for the report
    metrics: List[MetricSpec]       # Aggregate expressions
    group_by: List[str]             # Table-qualified column names
    filters: List[FilterSpec]       # WHERE predicates
    limit: Optional[int]            # Row limit
    base_sql: Optional[str]         # Subquery override
    raw_sql: Optional[str]          # Complete SQL override
    tables: List[str]               # Auto-populated by planner
    join_edges: List[str]           # Auto-populated by planner
```

The specification supports three execution modes based on which fields are populated:

| Mode | Condition | Behavior |
|---|---|---|
| raw_sql | raw_sql is not None | Execute raw_sql directly, ignore other fields |
| base_sql | raw_sql is None, base_sql is not None | Wrap base_sql as subquery, apply metrics/filters on top |
| auto | Both are None | Infer tables from specifications, build joins automatically |

## 5.2 Schema Graph (schema_graph.py)

The schema graph provides a graph-based representation of the database schema, enabling automatic join path discovery.

### 5.2.1 FKEdge

Represents an undirected edge between two tables connected by a foreign key relationship.

```
@dataclass
class FKEdge:
    table_a: str  # First table in the relationship
    table_b: str  # Second table in the relationship
```

The edge is undirected because the join can be traversed in either direction.

### 5.2.2 SchemaGraph

The main graph class wrapping SQLAlchemy metadata.

```
class SchemaGraph:
    def __init__(self, metadata: MetaData):
        self.metadata = metadata
        self._adj: Dict[str, Set[str]] = defaultdict(set)
        self._build_graph()

    def tables(self) -> List[str]: ...
    def neighbors(self, table: str) -> Set[str]: ...
    def build_join_forest(self, needed_tables: Set[str]) -> List[FKEdge]: ...
```

Key operations:

- tables(): Returns all table names in the schema
- neighbors(table): Returns tables directly connected via FK to the given table
- build_join_forest(needed_tables): Computes a minimal spanning tree connecting all needed tables

### 5.2.3 Join Forest Algorithm

The `build_join_forest` method uses breadth-first search to find a minimal set of edges connecting all required tables:

```
def build_join_forest(self, needed_tables: Set[str]) -> List[FKEdge]:
    # Pick arbitrary starting table
    start = next(iter(needed_tables))
    visited = {start}
    parent: Dict[str, str] = {}
    queue = deque([start])
    remaining = set(needed_tables) - {start}

    # BFS restricted to needed_tables
    while queue and remaining:
        current = queue.popleft()
        for neighbor in self.neighbors(current):
            if neighbor not in needed_tables:
                continue
            if neighbor not in visited:
                visited.add(neighbor)
                parent[neighbor] = current
                queue.append(neighbor)
                remaining.discard(neighbor)

    if remaining:
        raise ValueError(f'Tables not connected: {remaining}')

    # Reconstruct edges from parent pointers
    return [FKEdge(par, child) for child, par in parent.items()]
```

## 5.3 HTN Core (htn_core.py)

The HTN core module implements a minimal but complete HTN planner.

### 5.3.1 Task

Represents a task (either primitive or compound) to be accomplished.

```
@dataclass
class Task:
    name: str                              # Task identifier
    args: Tuple[Any, ...] = field(...)     # Positional arguments
    kwargs: Dict[str, Any] = field(...)    # Keyword arguments
```

### 5.3.2 Method

Defines a decomposition rule for compound tasks.

```
@dataclass
class Method:
    task_name: str                                     # Compound task this
applies to
    condition: Callable[[Any, Task], bool]             # Applicability predicate
    decompose: Callable[[Any, Task], List[Task]]       # Produces subtasks
    name: str = ''                                     # Human-readable
identifier
```

### 5.3.3 PrimitiveOp

Defines an executable primitive action.

```
@dataclass
class PrimitiveOp:
    task_name: str                         # Primitive task this implements
    apply: Callable[[Any, Task], Any]      # State transformation function
```

### 5.3.4 HTNPlanner

The planner orchestrates task decomposition and execution.

```
class HTNPlanner:
    def __init__(self, methods: List[Method], ops: List[PrimitiveOp]):
        self._methods_by_task = {}  # task_name → [methods]
        self._ops_by_task = {}      # task_name → primitive_op

    def plan_and_execute(self, state: Any, top_task: Task) -> Any:
        return self._execute_task(state, top_task)

    def _execute_task(self, state: Any, task: Task) -> Any:
        # 1. Check if primitive
        if task.name in self._ops_by_task:
            return self._ops_by_task[task.name].apply(state, task)

        # 2. Try methods in order
        for method in self._methods_by_task.get(task.name, []):
            if method.condition(state, task):
                branch_state = copy(state)
                subtasks = method.decompose(branch_state, task)
                for subtask in subtasks:
                    branch_state = self._execute_task(branch_state, subtask)
                return branch_state

        raise PlanningFailure(f'No methods for {task.name}')
```

## 5.4 Planning State (planning_state.py)

The mutable state object passed through the planning process.

```
@dataclass
class PlanningState:
    engine: Engine                # SQLAlchemy database connection
    schema_graph: SchemaGraph     # Schema metadata graph
    spec: ReportSpec              # Input specification

    execution_mode: Optional[str] = None      # 'raw_sql', 'base_sql', 'auto'
    inferred_tables: Set[str] = field(...)    # Tables needed for query
    join_forest: Any = None                   # List[FKEdge] or similar
    sql_ast: Any = None                       # SQLAlchemy Select object
    sql_text: Optional[str] = None            # Generated SQL string
    result_rows: Optional[List[Dict]] = None  # Query results
    plan_trace: List[PlanStep] = field(...)   # Debug trace
    scratch: Dict[str, Any] = field(...)      # Agent-specific data
```

The state is progressively populated as planning proceeds:

1. Initial: engine, schema_graph, spec are set
2. After ChooseExecutionMode: execution_mode is set
3. After InferTablesFromSpec: inferred_tables is populated
4. After FindJoinForest: join_forest contains FK edges
5. After BuildSqlFromPlan: sql_ast or sql_text is set
6. After ExecutePlannedSql: result_rows contains the query results

## 5.5 Planning Domain (planning_domain_basic.py)

Defines the core planning methods and primitives for HTNQL.

### 5.5.1 Task Decomposition Hierarchy

The planning domain defines a hierarchy of tasks:

```
AnswerReport
├── ChooseExecutionMode
│   ├── (raw_sql mode)
│   ├── (base_sql mode)
│   └── (auto mode)
├── PlanExecution
│   ├── PlanRawSql
│   ├── PlanBaseSql
│   └── PlanAutoSql
│       ├── ValidateSpecStructurally
│       ├── InferTablesFromSpec
│       ├── AnalyzeComplexity
│       ├── FindJoinForest
│       └── BuildSqlFromPlan
└── ExecutePlannedSql
```

### 5.5.2 Method Definitions

Each compound task has one or more methods. The planner tries methods in order until one succeeds.

**AnswerReport**: Single method that decomposes into ChooseExecutionMode → PlanExecution → ExecutePlannedSql

**ChooseExecutionMode**: Three methods checking for raw_sql, base_sql, or auto mode

**PlanExecution**: Three methods branching on execution_mode

**PlanAutoSql**: Single method implementing the full auto-planning pipeline

### 5.5.3 Primitive Operations

The following primitives are defined:

| Primitive | Purpose | State Modifications |
|---|---|---|
| ValidateSpecStructurally | Check spec has valid structure | Raises on invalid spec |
| InferTablesFromSpec | Extract table names from columns | Sets inferred_tables |
| AnalyzeComplexity | Check table count is reasonable | Raises if too complex |
| FindJoinForest | Compute join path via schema graph | Sets join_forest |
| BuildSqlFromPlan | Generate SQLAlchemy or text SQL | Sets sql_ast or sql_text |
| ExecutePlannedSql | Run query and collect results | Sets result_rows |
| PlanRawSql | Use raw_sql directly | Sets sql_text |
| PlanBaseSql | Wrap base_sql as subquery | Sets sql_text |

## 5.6 Agent DSL (agent_dsl.py)

The agent DSL enables defining planning behaviors through configuration dictionaries rather than code.

### 5.6.1 Configuration Structure

An agent configuration has the following structure:

```
{
    'tasks': {
        'TaskName': {
            'methods': [
                {
                    'name': 'MethodName',
                    'when': [
                        {'field': 'path.to.field', 'op': 'eq', 'value':
expected}
                    ],
                    'steps': [
                        {'task': 'SubtaskName'},
                        {'primitive': 'PrimitiveName'}
                    ]
                }
            ]
        }
    }
}
```

### 5.6.2 Condition Operators

The following condition operators are supported:

| Operator | Meaning | Example |
|---|---|---|
| eq | Equals | {'op': 'eq', 'value': 'auto'} |
| neq | Not equals | {'op': 'neq', 'value': None} |
| is_null | Is None | {'op': 'is_null'} |
| is_not_null | Is not None | {'op': 'is_not_null'} |
| is_true | Boolean true | {'op': 'is_true'} |
| is_false | Boolean false | {'op': 'is_false'} |
| size_gte | Collection size ≥ n | {'op': 'size_gte', 'value': 1} |
| size_lte | Collection size ≤ n | {'op': 'size_lte', 'value': 6} |

## 5.7 Built-in Agents (builtin_agents.py)

HTNQL ships with two pre-defined agents:

### 5.7.1 strict_joins

The default agent that requires all tables to be connected via foreign keys.

```
'strict_joins': {
    'tasks': {
        'FindJoinForest': {
            'methods': [{
                'name': 'StrictFK',
                'when': [{'field': 'inferred_tables', 'op': 'size_gte', 'value':
1}],
                'steps': [{'primitive': 'FindJoinForest.StrictFK'}]
            }]
        },
        'BuildSqlFromPlan': {
            'methods': [{
                'name': 'SqlAlchemyOnly',
                'when': [],
                'steps': [{'primitive': 'BuildSql.SqlAlchemy'}]
            }]
        }
    }
}
```

### 5.7.2 heuristic_joins

An alternative agent that includes fallback strategies:

- Attempts strict FK joins first
- Falls back to heuristic joins if strict fails
- Can generate raw SQL if SQLAlchemy building fails

## 5.8 Shape Suggestion (shape_suggestion.py)

Provides intelligent suggestions for query "shapes"—recommended combinations of tables and joins.

### 5.8.1 ShapeIntent

Describes what kind of query the user is interested in:

```
@dataclass
class ShapeIntent:
    description: str = ''
    focus_entities: List[str] = field(...)      # Entity names to focus on
    time_grain: Optional[str] = None            # 'day', 'month', 'year'
    metric_hints: List[str] = field(...)        # Keywords for metrics
    include_tables: List[str] = field(...)      # Tables to include
    exclude_tables: List[str] = field(...)      # Tables to exclude
```

### 5.8.2 CandidateShape

A suggested query structure:

```
@dataclass
class CandidateShape:
    id: str                           # Unique identifier
    description: str                  # Human-readable description
    base_sql: str                     # Suggested base SQL with joins
    tables: List[str]                 # Tables involved
    columns: List[ColumnRole]         # Columns with inferred roles
    confidence: float                 # 0.0 to 1.0 confidence score
    notes: Dict[str, str] = field(...)
```

### 5.8.3 Column Role Inference

The system automatically infers column roles:

- dimension: String or foreign key columns (for GROUP BY)
- measure: Numeric columns (for aggregation)
- time: Date/datetime columns (for time-based analysis)

### 5.8.4 Ranking Algorithm

Shapes are ranked by a scoring formula:

score = numeric_columns + fk_degree + name_match_bonus

where:

- numeric_columns: Count of numeric columns (potential measures)
- fk_degree: Number of foreign key relationships (joins to dimensions)
- name_match_bonus: +2 for each focus entity or metric hint found in table name

## 5.9 Query Engine (query_engine.py)

The main orchestration layer that ties all components together.

### 5.9.1 Initialization

The QueryEngine is initialized with a database connection, schema graph, and optional agent selection:

```
class QueryEngine:
    def __init__(
        self,
        engine: Engine,
        schema_graph: SchemaGraph,
        agent: str = 'strict_joins',
        agents_config: Optional[Dict] = None
    ):
        self.engine = engine
        self.schema_graph = schema_graph

        # Build planning domain from base + agent config
        base_methods, base_ops = build_basic_planning_domain()
        agent_methods = build_methods_from_agent_config(agents_config[agent])
        agent_ops = collect_primitives_for_agent(agents_config[agent],
PRIMITIVE_REGISTRY)

        # Create planner with combined methods/ops
        self._planner = HTNPlanner(
            methods=base_methods + agent_methods,
            ops=base_ops + agent_ops
        )
```

### 5.9.2 Running Reports

Two methods are provided for executing reports:

```
def run_report(self, spec: ReportSpec) -> List[Dict[str, Any]]:
    '''Execute report and return results.'''
    state = PlanningState(engine=self.engine, schema_graph=self.schema_graph,
spec=spec)
    final_state = self._planner.plan_and_execute(state,
Task(name='AnswerReport'))
    return final_state.result_rows or []

def run_report_with_trace(self, spec: ReportSpec) -> Tuple[List[Dict],
List[PlanStep]]:
    '''Execute report and return results plus planning trace.'''
    state = PlanningState(engine=self.engine, schema_graph=self.schema_graph,
spec=spec)
    final_state = self._planner.plan_and_execute(state,
Task(name='AnswerReport'))
    return final_state.result_rows or [], final_state.plan_trace
```

# PART III

GUI Application

# 6. PySide6 Visual Query Builder

HTNQL includes a PySide6-based desktop application that provides a visual interface for building and executing reports. This chapter documents the GUI architecture and usage.

## 6.1 Application Architecture

The GUI follows a clean separation between the session layer (business logic) and the presentation layer (Qt widgets).

### 6.1.1 HTNQLSession

The session class wraps HTNQL functionality for GUI consumption:

```
class HTNQLSession:
    def __init__(self, url: str):
        self.engine = create_engine(url)
        md = MetaData()
        md.reflect(bind=self.engine)
        self.schema_graph = SchemaGraph(md)
        self.qe = QueryEngine(self.engine, self.schema_graph)

    def list_tables(self) -> List[Dict]: ...
    def get_columns_for_table(self, table: str) -> List[Dict]: ...
    def suggest_shapes_for_table(self, table: str) -> List[Dict]: ...
    def run_report(self, spec_dict: dict) -> Dict: ...
```

### 6.1.2 Widget Hierarchy

The main window contains the following widget hierarchy:

```
MainWindow
├── QMenuBar
│   └── File menu (Connect, Exit)
├── QStatusBar
└── QSplitter (horizontal)
    ├── SchemaBrowser (left panel)
    │   └── QListWidget (table list)
    └── QSplitter (vertical, right side)
        ├── QueryBuilder (top)
        │   ├── Base table + Shape selector
        │   ├── Metrics table
        │   ├── Group-by table
        │   ├── Filters table
        │   └── Limit spin + Run button
        └── ResultView (bottom)
            └── QTabWidget
                ├── Results tab (QTableWidget)
                └── Plan/Debug tab (QPlainTextEdit)
```

## 6.2 User Interface Components

### 6.2.1 Connection Dialog

The connection dialog allows users to select a SQLite database file. The dialog builds a SQLAlchemy URL from the selected file path.

Fields:

- Backend selector (currently SQLite only, extensible to other backends)
- File path input with Browse button

### 6.2.2 Schema Browser

The left panel displays all tables discovered in the connected database. Clicking a table:

1. Sets it as the base table in the query builder
2. Populates the Group By panel with the table's columns
3. Triggers shape suggestion for the selected table

### 6.2.3 Query Builder

The query builder panel contains controls for constructing a report specification:

**Base Table:** Read-only field showing the currently selected table

**Shape Selector:** Combo box populated with suggested query shapes

**Metrics Table:** Editable table with Expression and Alias columns. Users can add/remove rows.

**Group By Table:** Checkbox list of columns from the base table

**Filters Table:** Three-column table with Column (combo), Operator (combo), and Value (text)

**Limit Spinner:** Integer input for row limit (default 1000)

**Run Button:** Executes the constructed query

### 6.2.4 Result View

The bottom panel contains a tabbed interface:

**Results Tab:** QTableWidget displaying query results with column headers

**Plan/Debug Tab:** QPlainTextEdit showing the HTN planning trace

## 6.3 Query Execution Flow

When the user clicks Run, the following sequence occurs:

1. QueryBuilder.build_spec_dict() constructs a dictionary representation of the report
2. The dictionary includes metrics, group_by, filters, limit, and base_sql
3. MainWindow passes the dictionary to HTNQLSession.run_report()
4. The session converts the dict to a ReportSpec and calls QueryEngine.run_report_with_trace()
5. Results are returned as rows (list of lists), headers (column names), and trace (planning steps)
6. ResultView.set_rows() populates the results table
7. ResultView.set_debug_text() displays the planning trace

Specification dictionary format:

```
{
    'name': 'gui_<base_table>',
    'metrics': [{'expr': 'COUNT(*)', 'alias': 'count'}, ...],
    'group_by': ['table.column', ...],
    'filters': [{'column': 'table.col', 'op': '=', 'value': 'X'}, ...],
    'limit': 1000,
    'base_sql': 'SELECT * FROM <base_table>'
}
```

## 6.4 Error Handling

The GUI handles errors at multiple levels:

- Connection Errors: Displayed via QMessageBox when database connection fails
- Schema Errors: Shown when schema reflection encounters problems
- Query Errors: SQL or planning errors are caught and displayed to the user
- No Database: Warning shown if user attempts to run without connecting first

# PART IV

Demonstration Databases

# 7. Demonstration Databases

HTNQL includes three demonstration databases that showcase different use cases and schema patterns. Each database is self-contained with schema definition, population script, and sample queries.

## 7.1 Ledger Demo (Double-Entry Accounting)

The ledger demo implements a double-entry bookkeeping system, demonstrating HTNQL's ability to handle financial reporting requirements.

### 7.1.1 Schema

```
accounts
├── id (PK)
├── code (unique account code)
├── name
└── type (ASSET, LIABILITY, EQUITY, REVENUE, EXPENSE)

journals
├── id (PK)
└── name (unique journal name)

transactions
├── id (PK)
├── journal_id (FK → journals)
├── txn_date (ISO date)
├── description
└── ref (reference number)

entries
├── id (PK)
├── transaction_id (FK → transactions)
├── account_id (FK → accounts)
└── amount_cents (positive = debit, negative = credit)
```

### 7.1.2 Relationship Diagram

```
journals ←──┐
            │
accounts ←──┼── entries ──→ transactions
            │
            └── (FK relationships)
```

### 7.1.3 Sample Reports

**Trial Balance:** Sum of all entries grouped by account

```
ReportSpec(
    name='trial_balance',
    metrics=[MetricSpec('SUM(entries.amount_cents)', 'balance_cents')],
    group_by=['accounts.code', 'accounts.name', 'accounts.type'],
    filters=[]
)
```

**Monthly Revenue:** Revenue accounts aggregated by month

```
ReportSpec(
    name='monthly_revenue',
    metrics=[MetricSpec('SUM(entries.amount_cents)', 'revenue_cents')],
    group_by=['accounts.type'],
    filters=[FilterSpec('accounts.type', '=', 'REVENUE')]
)
```

## 7.2 Issues Demo (Issue Tracking)

The issues demo models a project issue tracking system, suitable for bug tracking and task management reporting.

### 7.2.1 Schema

```
projects
├── id (PK)
├── name
└── description

issues
├── id (PK)
├── project_id (FK → projects)
├── title
├── description
├── status (OPEN, IN_PROGRESS, RESOLVED, CLOSED)
├── priority (LOW, MEDIUM, HIGH, CRITICAL)
├── created_at (datetime)
└── closed_at (datetime, nullable)
```

### 7.2.2 Sample Reports

**Issues by Status:** Count of issues grouped by status

```
ReportSpec(
    name='issues_by_status',
    metrics=[MetricSpec('COUNT(*)', 'issue_count')],
    group_by=['issues.status'],
    filters=[]
)
```

**Average Resolution Time:** Days to close issues (where closed)

```
ReportSpec(
    name='avg_resolution_time',
    metrics=[MetricSpec(
        'AVG(julianday(closed_at) - julianday(created_at))',
        'avg_days_to_close'
    )],
    group_by=['issues.priority'],
    filters=[FilterSpec('issues.closed_at', '!=', '')]
)
```

**Open Issues by Project:** Count of open issues per project

```
ReportSpec(
    name='open_by_project',
    metrics=[MetricSpec('COUNT(*)', 'open_count')],
    group_by=['projects.name'],
    filters=[FilterSpec('issues.status', 'IN', "('OPEN','IN_PROGRESS')")]
)
```

## 7.3 Airbnb Demo (Rental Marketplace)

The Airbnb demo models a vacation rental marketplace with hosts, listings, and bookings.

### 7.3.1 Schema

```
hosts
├── id (PK)
├── name
├── email
└── joined_at (datetime)

listings
├── id (PK)
├── host_id (FK → hosts)
├── title
├── description
├── property_type (apartment, house, villa, etc.)
├── city
├── price_per_night_cents
└── created_at (datetime)

bookings
├── id (PK)
├── listing_id (FK → listings)
├── guest_name
├── check_in_date
├── check_out_date
├── total_price_cents
├── status (PENDING, CONFIRMED, CANCELLED, COMPLETED)
└── created_at (datetime)
```

### 7.3.2 Relationship Diagram

```
hosts ←── listings ←── bookings

Cardinality:
  host (1) → listings (many)
  listing (1) → bookings (many)
```

### 7.3.3 Sample Reports

**Revenue by Status:** Total booking revenue grouped by status

```
ReportSpec(
    name='revenue_by_status',
    metrics=[MetricSpec('SUM(bookings.total_price_cents)', 'revenue_cents')],
    group_by=['bookings.status'],
    filters=[]
)
```

**Revenue by City:** Revenue aggregated by listing city (requires join)

```
ReportSpec(
    name='revenue_by_city',
    metrics=[MetricSpec('SUM(bookings.total_price_cents)', 'revenue_cents')],
    group_by=['listings.city'],
    filters=[FilterSpec('bookings.status', '=', 'CONFIRMED')]
)
```

**Top Hosts by Booking Count:** Number of bookings per host

```
ReportSpec(
    name='top_hosts',
    metrics=[MetricSpec('COUNT(*)', 'booking_count')],
    group_by=['hosts.name'],
    filters=[],
    limit=10
)
```

# PART V

Advanced Topics and Extensions

# 8. Extending HTNQL

This chapter covers advanced topics for developers who wish to extend or customize HTNQL.

## 8.1 Creating Custom Agents

New planning strategies can be added by defining agent configurations:

```
CUSTOM_AGENT = {
    'tasks': {
        'FindJoinForest': {
            'methods': [
                {
                    'name': 'OptimizedJoins',
                    'when': [
                        {'field': 'inferred_tables', 'op': 'size_lte', 'value':
3}
                    ],
                    'steps': [
                        {'primitive': 'FindJoinForest.StrictFK'}
                    ]
                },
                {
                    'name': 'LargeQueryFallback',
                    'when': [],
                    'steps': [
                        {'primitive': 'FindJoinForest.Heuristic'}
                    ]
                }
            ]
        }
    }
}

# Use custom agent
qe = QueryEngine(engine, sg, agent='custom', agents_config={'custom':
CUSTOM_AGENT})
```

## 8.2 Adding New Primitives

Custom primitive operations can be registered:

```
from htnql.htn_core import PrimitiveOp
from htnql.planning_primitives import PRIMITIVE_REGISTRY

def _apply_custom_optimization(state, task):
    # Implement custom logic
    # Modify state as needed
    return state
```

```
PRIMITIVE_REGISTRY['CustomOptimization'] = PrimitiveOp(
    task_name='CustomOptimization',
    apply=_apply_custom_optimization
)
```

## 8.3 Serialization and Saved Reports

ReportSpec instances can be serialized to JSON for persistence:

```
import json
from dataclasses import asdict

def spec_to_dict(spec: ReportSpec) -> dict:
    return {
        'name': spec.name,
        'metrics': [asdict(m) for m in spec.metrics],
        'group_by': spec.group_by,
        'filters': [asdict(f) for f in spec.filters],
        'limit': spec.limit,
        'base_sql': spec.base_sql,
        'raw_sql': spec.raw_sql,
    }

def spec_from_dict(d: dict) -> ReportSpec:
    return ReportSpec(
        name=d['name'],
        metrics=[MetricSpec(**m) for m in d['metrics']],
        group_by=d.get('group_by', []),
        filters=[FilterSpec(**f) for f in d.get('filters', [])],
        limit=d.get('limit'),
        base_sql=d.get('base_sql'),
        raw_sql=d.get('raw_sql'),
    )

# Save
with open('report.json', 'w') as f:
    json.dump(spec_to_dict(spec), f, indent=2)

# Load
with open('report.json') as f:
    spec = spec_from_dict(json.load(f))
```

## 8.4 Performance Considerations

### 8.4.1 Query Complexity Limits

HTNQL enforces a maximum of 6 tables per query by default. This can be adjusted:

```
# In planning_domain_basic.py
MAX_TABLES = 6  # Change this value
```

### 8.4.2 Schema Graph Caching

For large schemas, consider caching the SchemaGraph:

```
# Build once, reuse
sg = SchemaGraph(metadata)

# Use for multiple engines with same schema
qe1 = QueryEngine(engine1, sg)
qe2 = QueryEngine(engine2, sg)
```

### 8.4.3 Connection Pooling

For high-throughput scenarios, use SQLAlchemy connection pooling:

```
from sqlalchemy.pool import QueuePool

engine = create_engine(
    'sqlite:///database.db',
    poolclass=QueuePool,
    pool_size=5,
    max_overflow=10
)
```

## 8.5 Testing and Debugging

### 8.5.1 Using the Planning Trace

The planning trace provides insight into the planner's decisions:

```
rows, trace = qe.run_report_with_trace(spec)

for step in trace:
    print(f'Task: {step.task.name}')
    print(f'Method: {step.method_name}')
    print('---')
```

### 8.5.2 Inspecting Generated SQL

Access the generated SQL from the planning state:

```
state = PlanningState(engine=engine, schema_graph=sg, spec=spec)
final_state = qe._planner.plan_and_execute(state, Task(name='AnswerReport'))

print('SQL Text:', final_state.sql_text)
print('SQL AST:', final_state.sql_ast)
```

# 9. API Reference Summary

This chapter provides a quick reference to the main classes and functions in HTNQL.

## 9.1 Core Classes

| Class | Module | Purpose |
|---|---|---|
| ReportSpec | report_spec | Declarative report specification |
| MetricSpec | report_spec | Single metric/aggregate definition |
| FilterSpec | report_spec | WHERE clause predicate |
| SchemaGraph | schema_graph | Graph of tables and FK relationships |
| FKEdge | schema_graph | Edge between two related tables |
| QueryEngine | query_engine | Main entry point for query execution |
| HTNPlanner | htn_core | HTN planning algorithm implementation |
| Task | htn_core | Task to be accomplished |
| Method | htn_core | Decomposition rule for compound task |
| PrimitiveOp | htn_core | Executable primitive action |
| PlanningState | planning_state | Mutable state during planning |
| ShapeIntent | shape_suggestion | Query shape request |
| CandidateShape | shape_suggestion | Suggested query structure |

## 9.2 Key Functions

| Function | Module | Purpose |
|---|---|---|
| build_basic_planning_domain() | planning_domain_basic | Returns base methods and primitives |
| build_methods_from_agent_config() | agent_dsl | Converts agent config to Method objects |
| collect_primitives_for_agent() | agent_dsl | Extracts primitives needed by agent |
| suggest_shapes() | shape_suggestion | Returns ranked CandidateShape list |

## 9.3 GUI Classes

| Class | Purpose |
|---|---|
| HTNQLSession | Session wrapper bridging GUI and library |
| MainWindow | Top-level application window |
| ConnectionDialog | Database connection dialog |
| SchemaBrowser | Left panel table list |
| QueryBuilder | Report specification form |
| ResultView | Results and debug output display |

# 10. Conclusion

HTNQL represents an innovative application of AI planning techniques to the domain of database query generation. By treating SQL construction as a hierarchical task network planning problem, HTNQL achieves several notable benefits:

- Declarative Interface: Users specify what data they need, not how to join tables
- Automatic Join Inference: Foreign key relationships guide join path discovery
- Extensible Architecture: New planning strategies can be added through configuration
- Debugging Support: Planning traces provide insight into query construction
- Graceful Fallback: Escape hatches (base_sql, raw_sql) when automatic planning is insufficient

## 10.1 Theoretical Contributions

This work demonstrates that HTN planning—a technique traditionally applied to robotics, game AI, and autonomous systems—can be effectively applied to the domain of database query generation. The key insight is that query construction naturally decomposes into a hierarchy of tasks: choosing an execution strategy, inferring required tables, computing join paths, and generating SQL syntax.

## 10.2 Practical Applications

HTNQL is suitable for several practical applications:

- Business Intelligence: Ad-hoc reporting over relational databases
- Data Exploration: Interactive query building for data analysts
- Application Development: Embedding declarative query capabilities in applications
- Education: Teaching database concepts through visual query building

## 10.3 Future Directions

Several avenues for future development exist:

1. Query Optimization: Integrate cost-based optimization into the planning process
2. Natural Language Interface: Parse English queries into ReportSpec objects
3. Multi-Database Support: Extend beyond SQLite to PostgreSQL, MySQL, etc.
4. Incremental Planning: Support for query modification and refinement
5. Learning from Feedback: Use execution statistics to improve planning heuristics

HTNQL provides a solid foundation for exploring these directions while remaining immediately useful for practical database reporting tasks.

# APPENDICES

## Appendix A: Installation Guide

### A.1 Prerequisites

- Python 3.10 or higher
- pip package manager
- SQLite (included with Python)

### A.2 Installation Steps

```
# Clone repository
git clone <repository-url>
cd htnql

# Create virtual environment
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate

# Install dependencies
pip install sqlalchemy
pip install pyside6  # For GUI

# Install HTNQL in development mode
pip install -e .
```

### A.3 Running the GUI

```
python gui.py
```

### A.4 Populating Demo Databases

```
cd htnql_demos/ledger_demo
python populate_ledger_demo.py

cd ../issues_demo
python populate_issues_demo.py

cd ../airbnb_demo
python populate_airbnb_demo.py
```

## Appendix B: Complete Example Session

This appendix demonstrates a complete HTNQL session from connection to query execution.

```
from sqlalchemy import create_engine, MetaData
from htnql.schema_graph import SchemaGraph
from htnql.query_engine import QueryEngine
from htnql.report_spec import ReportSpec, MetricSpec, FilterSpec

# 1. Connect to database
engine = create_engine('sqlite:///htnql_demos/ledger_demo/ledger.db')

# 2. Reflect schema
metadata = MetaData()
metadata.reflect(bind=engine)

# 3. Build schema graph
sg = SchemaGraph(metadata)

# 4. Inspect available tables
print('Tables:', sg.tables())
# Output: ['accounts', 'entries', 'journals', 'transactions']

# 5. Create query engine
qe = QueryEngine(engine, sg)

# 6. Define report specification
spec = ReportSpec(
    name='trial_balance',
    metrics=[
        MetricSpec(expr='SUM(entries.amount_cents)', alias='balance_cents')
    ],
    group_by=[
        'accounts.code',
        'accounts.name',
        'accounts.type'
    ],
    filters=[],
    limit=100
)

# 7. Execute with trace
rows, trace = qe.run_report_with_trace(spec)

# 8. Display results
for row in rows[:5]:
    print(row)

# 9. Inspect planning trace
print('\nPlanning Trace:')
for step in trace:
    print(f'  {step.task.name} -> {step.method_name}')
```

## Appendix C: Glossary

**Agent**: A configuration defining planning behavior, including method selection and primitive operations.

**Base SQL**: A user-provided SQL query that serves as a subquery for further metric/filter application.

**Compound Task**: A task that must be decomposed into subtasks before execution.

**Filter**: A WHERE clause predicate restricting query results.

**Foreign Key (FK)**: A database constraint linking columns between tables.

**HTN (Hierarchical Task Network)**: A planning formalism based on task decomposition.

**Join Forest**: A set of edges connecting tables needed for a query.

**Metric**: An aggregate expression in the SELECT list (e.g., SUM, COUNT, AVG).

**Method**: A rule defining how to decompose a compound task into subtasks.

**Planning State**: The mutable state passed through the HTN planner.

**Primitive Task**: A task that can be executed directly without decomposition.

**Raw SQL**: User-provided SQL executed directly, bypassing the planner.

**ReportSpec**: The declarative specification of a report's requirements.

**Schema Graph**: A graph representation of database tables and their FK relationships.

**Shape**: A suggested query structure combining tables, joins, and column roles.

*— End of Document —*