Core Primitives: The Foundation of Numerical Inference

Before we get to complex relational inference, we need two robust, high-performance primitives.

1. The Prime Number Service

This is a critical, centralized helper service responsible for dispensing unique prime numbers for every new `ID` created.

- **Algorithm: Sieve and Dispense**
    - **Initialization**: On startup, the service pre-computes a large number of primes using the Sieve of Eratosthenes and stores them in a persistent, ordered data structure. A Redis Sorted Set is an excellent choice, where the score and value are both the prime number itself. This allows for efficient querying of ranges.
    - **Tracking**: A separate Redis key, let's call it `last_prime_dispensed`, stores the last prime number that was handed out.
    - Dispensing (Atomic Operation): When a new ID needs a prime, the service performs an atomic operation (e.g., a short Lua script in Redis) that:
      a. Reads the last_prime_dispensed.
      b. Queries the Redis Sorted Set for the next prime number immediately following it (ZRANGEBYSCORE ... LIMIT 1).
      c. Atomically updates last_prime_dispensed to this new prime.
      d. Returns the new prime to the caller.
    - **Implementation Note**: Using `java.math.BigInteger` is essential for all prime number calculations, as the products can quickly exceed the capacity of a standard `long`.

2. The Contextual Prime Product Embedding (CPPE) Calculator

This is a core function, likely within the **Index Service**, that calculates the embedding for an entity within a specific relational context.

- **Algorithm: `calculateCPPE(entityURI, contextPredicateURI)`**
    1. **Input**: The URI of the entity (e.g., `id:Google`) and the URI of the predicate that defines the context (e.g., `:worksFor`).
    2. **SPARQL Query**: Construct a SPARQL query to find all entities related to `entityURI` via the `contextPredicateURI`. The query needs to determine if `entityURI` is the subject or object to find the "other side" of the relation.
    3. Code snippet

```
# Simplified query if id:Google is the object
SELECT ?subjectPrime WHERE {
  ?subject <app:hasPrimeID> ?subjectPrime .
  ?subject <contextPredicateURI> <entityURI> .
}
```

4.
5.
6. **Prime Retrieval**: The query returns a list of the `primeID`s of all related entities (e.g., the primes for all employees of Google).
7. **Product Calculation**: Initialize a `BigInteger` to `1`. Iterate through the list of retrieved primes and multiply them together using `bigInteger.multiply()`.
8. **Return**: The final `BigInteger` product is the CPPE.

Advanced Relational Inference Algorithms

Now, let's build upon these primitives to perform complex inference.

1. Attribute Closure Inference (e.g., `knowsLanguage`)

This is about inferring a new, direct relationship from a recurring two-step path.

- **Schema Inference (Statistical Path Analysis)**
  - **Goal**: To discover the *rule* `(:Developer)-[:worksOn]->(:Project)-[:usesLanguage]->(:Language) ==> (:Developer)-[:knowsLanguage]->(:Language)`.
  - **Data Structure**: A persistent hash map, `PathFrequencyMap`, that stores `Key: (predicate1_URI, predicate2_URI)` and `Value: FrequencyCount (Integer)`.
  - **Algorithm (A periodic batch job in the Alignment Service)**:
    1. Execute a SPARQL query to find all unique pairs of connected predicates in the graph: `SELECT DISTINCT ?p1 ?p2 WHERE { ?a ?p1 ?b . ?b ?p2 ?c . }`.
    2. For each `(?p1, ?p2)` pair returned, increment its count in the `PathFrequencyMap`.
    3. After processing, iterate through the map. If any `FrequencyCount` for a pair `(P1, P2)` exceeds a configurable threshold `T_path` (e.g., 100 occurrences), it becomes a strong candidate for a new relational schema.
    4. Materialization: The system materializes the rule itself using OWL's property chain axiom, which is a formal way of stating the rule: <:knowsLanguage> owl:propertyChainAxiom (<:worksOn> <:usesLanguage>) .
- **Instance Inference (Numerical Confirmation using CPPE & GCD)**

- **Goal**: To confirm if a *specific* developer, `dev:Alice`, knows a *specific* language, `lang:Java`.
- **Algorithm**:
    1. Calculate Developer's Project Embedding:
       CPPE_Alice_Projects = calculateCPPE(dev:Alice, :worksOn)
       This returns the prime product of all projects Alice works on. (prime(projA) * prime(projB)).
    2. Calculate Language's Project Embedding: This requires an inverse context lookup.
       CPPE_Java_Projects = calculateCPPE(lang:Java, :usesLanguage_inverse)
       This returns the prime product of all projects that use Java. (prime(projB) * prime(projC)).
    3. Find Shared Context:
       SharedProjectsPrimeProduct = GCD(CPPE_Alice_Projects, CPPE_Java_Projects)
       The java.math.BigInteger.gcd() method is used here. The result will be prime(projB).
    4. **Inference**: Since the `SharedProjectsPrimeProduct` is greater than 1, the inference is confirmed. Alice knows Java because they share at least one project context (`projB`). The magnitude of the GCD indicates the "strength" or "evidence" for this inferred link.

2. Transitivity Inference (e.g., `partOf`)

This is about determining if a relationship's nature is inherently transitive.

- **Property Inference (Statistical Confirmation)**
    - **Goal**: To determine if the `:partOf` property should be formally declared as transitive.
    - **Algorithm (A periodic analysis job)**:
        1. Select the candidate property `P` (e.g., `:partOf`).
        2. **Count Path-2 Instances**: Execute a SPARQL query to count all instances of the two-step path: `SELECT (COUNT(*) AS ?count) WHERE { ?a P ?b . ?b P ?c . }`. Store this as `N_path2`.
        3. **Count Closed Path-3 Instances**: Execute a second query to count how many of those path-2 instances also have the "shortcut" link: `SELECT (COUNT(*) AS ?count) WHERE { ?a P ?b . ?b P ?c . ?a P ?c . }`. Store this as `N_closed`.
        4. **Calculate Confidence Score**: `TransitivityConfidence = N_closed / N_path2`.

5. **Decision**: If `TransitivityConfidence` is above a high threshold (e.g., `0.99`), the system has strong statistical evidence that the property is transitive.
6. **Materialization**: The system asserts the schema axiom in the graph: `<:partOf> rdf:type owl:TransitiveProperty .`

- **Instance Inference**: Once the property is declared transitive, the OWL reasoner (like the one built into Jena) handles the instance-level inference automatically. A query for `?x :partOf :Car` will correctly return `:SparkPlug` even if only the `(:SparkPlug :partOf :Engine)` and `(:Engine :partOf :Car)` triples exist explicitly. This offloads the computational work from query time to the reasoner.

3. Symmetry Inference (e.g., `spouseOf`)

The algorithm is very similar to Transitivity Inference.

- **Property Inference (Statistical Confirmation)**
  1. Select candidate property `P` (e.g., `:spouseOf`).
  2. **Count Forward Instances**: `SELECT (COUNT(*) AS ?count) WHERE { ?a P ?b . }`. Store as `N_forward`.
  3. **Count Symmetric Instances**: `SELECT (COUNT(*) AS ?count) WHERE { ?a P ?b . ?b P ?a . }`. Store as `N_symmetric`.
  4. **Calculate Confidence**: `SymmetryConfidence = N_symmetric / N_forward`.
  5. **Decision**: If the confidence is near `1.0`, materialize the schema: `<:spouseOf> rdf:type owl:SymmetricProperty .`

Summary of Numerical & Algorithmic Benefits

This approach provides a powerful alternative to purely connectionist (e.g., LLM/vector) or purely logical (e.g., Prolog) methods by combining their strengths:

- **Explainability**: Every inference can be traced back to a specific set of shared prime factors, representing concrete shared relationships in the data. It's not a "black box".
- **Computational Efficiency**: For structural and relational reasoning, integer arithmetic (`BigInteger` multiplication and GCD) is vastly more efficient than high-dimensional vector math or complex logical unification.
- **Data-Driven Schema Evolution**: The system doesn't require an ontologist to define all relational rules upfront. It can discover them statistically from the data itself and then formalize them using OWL axioms.
- **Hybrid Reasoning**: This numerical approach doesn't preclude the use of LLMs. It simply reserves them for what they do best: understanding unstructured text and semantic

nuance. The structural reasoning is handled by this more efficient and deterministic engine.