Proposta progetto

Specifiche del Linguaggio GomorraSQL

Autore: Angelo Alberico
Target: LLVM IR

1. Introduzione

GomorraSQL è un linguaggio di query dichiarativo (DSL) ispirato a SQL, localizzato semanticamente e lessicalmente sul dialetto napoletano (ispirato alla serie Gomorra). Il compilatore è progettato per analizzare query di selezione, validarle semanticamente rispetto a schemi CSV, generare codice LLVM IR per l'esecuzione JIT e supportare operazioni JOIN.

Ispirazione: GomorraSQL

---

2. Specifiche Lessicali

Questa sezione definisce i lessemi validi del linguaggio. L'analisi lessicale è case-insensitive per le keyword.

2.1 Keyword Principali (DQL)

Token GomorraSQL	Corrispondente SQL	Descrizione
`ripigliammo`	`SELECT`	Inizio della proiezione dei dati
`mmiez 'a`	`FROM`	Specifica la sorgente dei dati
`tutto chillo ch'era 'o nuostro`	`*` (ALL)	Wildcard per selezionare tutte le colonne
`pesc e pesc`	`JOIN`	Clausola di unione tra tabelle
`arò`	`WHERE`	Introduttore della condizione di filtro

2.2 Operatori Logici e di Confronto

Gli operatori seguono la logica booleana standard ma utilizzano una sintassi localizzata.

- **Logici:**
- `e` (AND)
- `o` (OR)
- `nisciun` (NULL)
- `è` (IS) - usato per controlli null
- `nun è` (IS NOT)
- **Confronto:**
- `>`, `<`, `>=`, `<=`, `=`, `<>`, `!=`
- **Supporto completo:** Tutti gli operatori di confronto sono implementati e testati nel code generator LLVM.

2.3 Identificatori e Costanti

- **ID (CNAME):** Identificatori di tabelle e colonne.
- **STRING (ESCAPED_STRING):** Stringhe letterali racchiuse tra virgolette.
- **NUMBER (SIGNED_NUMBER):** Interi o decimali con segno.
- **BOOLEAN:** `true`, `false`.

---

3. Specifiche Sintattiche

La grammatica è definita in formato EBNF (target Lark).

3.1 Struttura del Programma (Query)

Una query valida consiste in un singolo statement di selezione.

```
select_stmt ::= SELECT_KW projection from_clause [where_clause]
```

3.2 Proiezioni e Sorgenti

La proiezione definisce quali colonne restituire.

```
projection  ::= ALL_COLS
              | column_list
column_list ::= column_ref ("," column_ref)*
from_clause ::= FROM_KW table_ref join_clause*
join_clause ::= JOIN_KW table_ref
```

> **Nota Semantica JOIN:**
> Il compilatore supporta JOIN tra due tabelle tramite prodotto cartesiano. Le tabelle vengono unite e le colonne duplicate vengono disambiguate automaticamente con suffisso numerico (_2, _3).
>
> **Limitazione attuale:** La sintassi grammaticale permette join multipli (`join_clause*`), ma l'implementazione corrente è ottimizzata per operazioni su singola tabella o JOIN tra due tabelle.

3.3 Clausola Where e Logica

La clausola `arò` definisce i filtri. La precedenza degli operatori vede `AND` (e) legare più forte di `OR` (o).

```
where_clause ::= WHERE_KW condition
condition    ::= logic_term (OR_KW logic_term)*
logic_term   ::= logic_factor (AND_KW logic_factor)*
logic_factor ::= comparison
               | null_check
               | "(" condition ")"
```

Supporto JOIN implementato:
Il compilatore supporta operazioni JOIN tra due tabelle tramite prodotto cartesiano. La sintassi permette di specificare multiple tabelle nella clausola FROM usando `pesc e pesc` (JOIN).

Esempio di Query con JOIN Valida

```
ripigliammo nome, ruolo
mmiez 'a "guaglioni.csv"
pesc e pesc "ruoli.csv"
arò nome = nome_2
```

> **Disambiguazione Colonne:**
> In caso di colonne con nome duplicato tra tabelle, il compilatore aggiunge automaticamente un suffisso `_2` , `_3` , etc. alle colonne successive.

Esempio di Query Valida con Condizioni Complesse

```
ripigliammo nome, cognome
mmiez 'a clan_savastano
```

```
arò eta > 18 e ruolo <> "boss"
```

Esempio Errato (Sintassi)

```
ripigliammo * arò eta > 18
mmiez 'a utenti
/* Errore: la clausola FROM (mmiez 'a) deve precedere WHERE (arò) */
```

---

4. Analisi Semantica

L'analisi semantica decora l'AST e verifica la coerenza logica della query rispetto allo schema dati (CSV).

4.1 Gestione dello Scoping e Schemi

A differenza dei linguaggi imperativi tradizionali che richiedono la costruzione di una symbol table dinamica, GomorraSQL costruisce il suo environment caricando gli header dei file CSV presenti nella directory `data`.

> **Regola Semantica (Esistenza Tabelle):**
> Ogni `table_ref` presente nella clausola `FROM` o `JOIN` deve corrispondere a un file CSV fisico esistente. Se il file non esiste, viene sollevato un `SemanticError`.

Gestione JOIN:
L'analizzatore semantico supporta la validazione di query con JOIN tra due tabelle. Durante l'analisi:

1. Carica gli schemi (header) di tutte le tabelle coinvolte
2. Disambigua automaticamente le colonne duplicate aggiungendo suffissi numerici
3. Valida che tutte le colonne referenziate esistano nell'insieme unificato

4.2 Type Checking e Validazione Colonne

Il compilatore esegue controlli di esistenza per ogni riferimento a colonna.

1. **Validazione Proiezione:**

- SE `projection` non è `*` (ALL_COLS),
- ALLORA per ogni `column_ref` in `column_list`:

- Verifica che la colonna esista nell'insieme delle colonne caricate dalle tabelle (incluse disambiguazioni da JOIN).
- ALTRIMENTI solleva errore: "Colonna non esiste nelle tabelle".

2. **Validazione WHERE:**

- Ogni identificatore usato in `comparison` o `null_check` deve appartenere allo schema delle tabelle in scope.
- **Supporto confronto colonna-colonna:** È ora possibile confrontare due colonne nella WHERE (es. `nome = nome_2`), supportando semantiche INNER JOIN.

4.3 Gestione dello Scoping

In GomorraSQL, la gestione dello scoping differisce dai linguaggi imperativi tradizionali in quanto non vi sono blocchi annidati o funzioni locali. Lo scope è determinato dal contesto di esecuzione della query, definito dalle tabelle specificate nella clausola `mmiez 'a` (FROM).

Costruzione della Tabella dei Simboli (Schema Environment)

L'analizzatore semantico costruisce dinamicamente un *environment* (equivalente alla tabella dei simboli) all'inizio dell'analisi della query.

1. **Identificazione Scope:** I costrutti che definiscono lo scope attivo sono la clausola `FROM` e le eventuali clausole `JOIN`.
2. **Caricamento Schemi:** Per ogni tabella menzionata, l'analizzatore accede al file CSV corrispondente e ne estrae l'header.
3. **Popolamento:** L'insieme `all_columns` funge da tabella dei simboli piatta, contenente tutti gli identificatori di colonna validi per la query corrente.

> **Regola di Scoping:**
> Un identificatore `ID` è visibile nello scope corrente se e solo se `ID` appartiene all'unione degli insiemi delle colonne delle tabelle caricate.
> `Scope = U { colonne(T) | T ∈ Tables(Query) }`

Se un identificatore utilizzato in `ripigliammo` (SELECT) o `arò` (WHERE) non è presente in questo insieme, viene sollevato un errore di "Colonna non trovata".

4.4 Type Checking e Sistema di Tipi

Il *Type Checking* in GomorraSQL verifica che le espressioni siano ben formate rispetto allo schema dati caricato. Poiché i CSV sono intrinsecamente non tipizzati (tutto è stringa o interpretabile come numero), il sistema applica una tipizzazione debole con controlli di esistenza rigorosi.

Regole di Inferenza e Controllo

Il sistema di tipi applica le seguenti regole di validazione (stile IF-THEN) durante la visita dell'AST:

1. Regola di Validazione Proiezione

Per ogni colonna `C` nella lista di proiezione:

- **IF** `C` è presente in `all_columns` (Environment)
- **THEN** la proiezione è semanticamente valida.
- **ELSE** Errore semantico: *"Colonna 'C' non esiste nelle tabelle"*.

2. Regola di Validazione Confronti (Comparison)

Dato un confronto `A OP B` (es. `eta > 18`):

- **IF** l'operando sinistro `A` è un identificatore presente in `all_columns`
- **AND** (l'operando destro `B` è un letterale **OR** `B` è un identificatore in `all_columns`)
- **THEN** il nodo `Comparison` è valido.
- **ELSE** Errore semantico: *"Colonna non esiste nella WHERE"*.

3. Regola di Validazione Null Check

Dato un controllo `A è nisciun` (IS NULL):

- **IF** `A` è presente in `all_columns`
- **THEN** il controllo è valido.
- **ELSE** Errore semantico.

---

5. Generazione Codice (LLVM IR)

Il backend genera codice LLVM IR utilizzando la libreria `llvmlite`. La strategia è compilare una funzione di filtraggio Just-In-Time (JIT) con fallback a esecuzione Python.

5.1 Strategia di Compilazione

Il compilatore non traduce l'intera logica di lettura file in IR, ma genera una funzione kernel ottimizzata per valutare singole righe.

Firma della funzione generata:

```
define i1 @evaluate_row(i32 %row_param)
```

Architettura Ibrida:

1. **LLVM IR Generation:** Genera sempre codice LLVM IR per la clausola WHERE
2. **JIT Compilation:** Tenta compilazione JIT nativa (fallisce su macOS ARM per mancanza target registration)
3. **Python Fallback:** Esegue la logica di filtraggio in Python interpretando la semantica dell'IR generato

5.2 Mapping dei Costrutti

Costrutto AST	Istruzione LLVM IR	Implementazione Python Fallback
`comparison (>)`	`icmp sgt`	`left_val > right_val`
`comparison (<)`	`icmp slt`	`left_val < right_val`
`comparison (>=)`	`icmp sge`	`left_val >= right_val`
`comparison (<=)`	`icmp sle`	`left_val <= right_val`
`comparison (=)`	`icmp eq`	`left_val == right_val`
`comparison (<>, !=)`	`icmp ne`	`left_val != right_val`
`AND` logic	`and i1`	`all(conditions)`
`OR` logic	`or i1`	`any(conditions)`
`where_clause`	`ret i1`	`return bool`

> **Nota Semantica:**
> Se `where_clause` è assente, la funzione generata contiene un singolo blocco base che ritorna la costante `1` (true).

Supporto JOIN nella Code Generation:

- Il codegen carica automaticamente multiple tabelle e genera il prodotto cartesiano
- Le colonne duplicate vengono disambiguate con suffissi numerici
- La WHERE può confrontare colonne di tabelle diverse (es. `nome = nome_2`)

5.3 Schema di Implementazione JIT con Fallback

1. **Parsing:** L'AST viene visitato dal `LLVMCodeGenerator` (implementa Visitor Pattern).
2. **IR Generation:**

- Viene creato un nuovo `ir.Module` per ogni query (evita duplicazioni di nomi)
- Una `ir.Function` viene generata con nome `evaluate_row`
- Il triple viene impostato al target nativo

3. **Building:** Un `ir.IRBuilder` popola i basic block con istruzioni `icmp`, `and`, `or`.
4. **Compilation Attempt:**
   - Viene tentata la creazione di un `ExecutionEngine` (MCJIT)
   - Su fallimento (es. target non registrato), viene loggato un warning
5. **Execution (Python Fallback):**
   - I dati CSV vengono caricati in Python
   - Per JOIN, viene generato il prodotto cartesiano
   - Ogni riga viene filtrata usando `_evaluate_condition_python()`
   - Il metodo Python interpreta ricorsivamente l'AST rispettando la semantica LLVM

5.4 Gestione Errori e Robustezza

Reset del Modulo:
Prima di ogni query, il modulo LLVM viene ricreato per evitare errori di duplicazione nomi di funzioni quando il compiler viene riutilizzato per multiple query.

```
self.module = ir.Module(name="gomorrasql_query")
self.module.triple = target.get_default_triple()
```