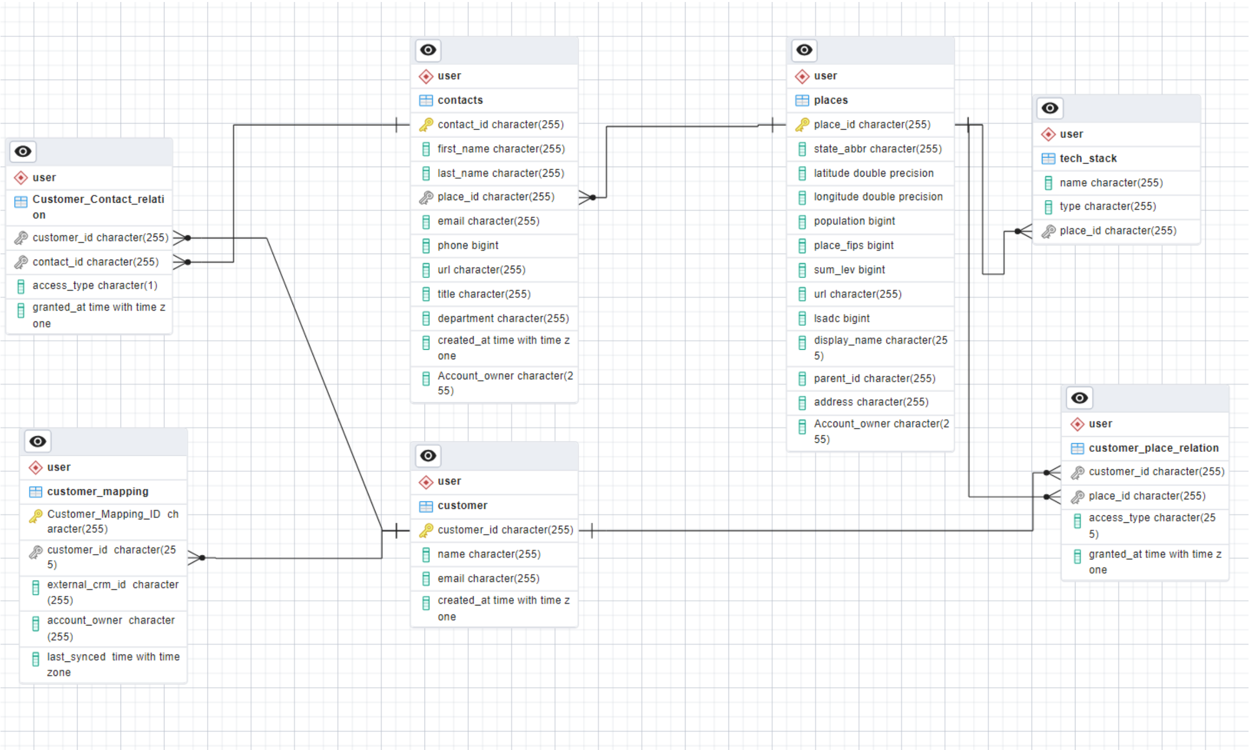
Pursuit Public Sector Data System


user
Customer_Contact_relation
customer_id character(255)
contact_id character(255)
access_type character(1)
granted_at time with time zone
user
contacts
contact_id character(255)
first_name character(255)
last_name character(255)
place_id character(255)
email character(255)
phone bigint
url character(255)
title character(255)
department character(255)
created_at time with time zone
Account_owner character(255)
user
places
place_id character(255)
state_abbr character(255)
latitude double precision
longitude double precision
population bigint
place_fips bigint
sum_lev bigint
url character(255)
lsadc bigint
display_name character(255)
parent_id character(255)
address character(255)
Account_owner character(255)
user
tech_stack
name character(255)
type character(255)
place_id character(255)
user
customer_mapping
Customer_Mapping_ID character(255)
customer_id character(255)
external_crm_id character (255)
account_owner character (255)
last_synced time with time zone
user
customer
customer_id character(255)
name character(255)
email character(255)
created_at time with time zone
user
customer_place_relation
customer_id character(255)
place_id character(255)
access_type character(255)
granted_at time with time zone

Part 1: System Design

Schema Overview

The schema consists of multiple tables designed to manage contacts, places, customer relationships, and technology stacks. The key entities and their relationships are outlined below:

Tables and Relationships

Customer

- Stores customer details such as ID, name, email, and creation.
- **Primary Key:** customer_id
- **Relationships:**
  - One-to-Many with customer_mapping
  - Many-to-Many with places via customer_place_relation
  - Many-to-Many with contacts via customer_contact_relation

Contacts

- Stores information about individuals, such as name, email, phone number, title, and department.
- **Primary Key:** contact_id
- **Relationships:**
    - One-to-Many with places
    - Many-to-Many with customers via customer_contact_relation

Places

- Represents organizations, locations, and governmental entities.
- **Primary Key:** place_id
- **Relationships:**
    - One-to-Many with contacts (a place can have multiple contacts)
    - One-to-Many with tech_stack (a place can have multiple technologies)
    - Many-to-Many with customers via customer_place_relation

Customer Contact Relation

- Manages the many-to-many relationship between customers and contacts.
- **Primary Keys:** customer_id, contact_id
- **Attributes:** access_type, granted_at

Customer Place Relation

- Manages the many-to-many relationship between customers and places.
- **Primary Keys:** customer_id, place_id
- **Attributes:** access_type, granted_at

Customer Mapping

- Links external CRM systems to customers for customer-specific mappings.
- **Primary Key:** Customer_Mapping_ID
- **Relationships:**
    - Many-to-One with customers (each mapping belongs to one customer)

Tech Stack

- Stores technology stacks associated with different places.
- **Relationships:**
    - Many-to-One with places (each place can have multiple technologies)

Indexing Strategy

To optimize performance for searching, filtering, and joining across large datasets(4.5 million contacts, 150k entities):

- **Primary Indexes**: Used on primary keys (customer_id, contact_id, place_id, etc.) optimized for fast lookups.
- **Foreign Key Indexes**: Speed up joins (e.g., place_id in contacts to places).

- **Composite Indexes**: Many queries filter based on multiple columns (e.g., title + department in contacts, or state_abbr + population in places). Adding composite indexes reduces query scan times significantly when filtering with multiple conditions.
- **Full-Text Indexes**: Enable efficient searches on text fields (title, emails, department in contacts). It optimizes name-based or keyword searches for large datasets.
- **Databricks Delta Lake Optimization**:
  - **ZORDER Indexing**: Optimized filtering and clustering (e.g., ZORDER BY place_id, contact_id, emails, Title etc.).
  - **Partitioning for faster scans:** Large tables should be partitioned to improve scan efficiency. For instance, places table to be Partitioned by state_abbr column and contacts table to be Partitioned by place_id column.
  - **Compaction (OPTIMIZE)**: Reduces small files, improving query performance.

```
CREATE INDEX idx_contact_name ON contacts(name);
CREATE INDEX idx_place_id ON contacts(place_id);
CREATE INDEX idx_customer_mapping ON customer_mapping(customer_id, external_crm_id);

OPTIMIZE contacts ZORDER BY (place_id, contact_id);
ALTER TABLE places SET TBLPROPERTIES ('delta.autoOptimize.optimizeWrite' = true);
```

Handling Customer-Specific Mappings

Key Tables and Their Roles

1. **customer_mapping Table**
   - Maps each customer_id to an external_crm_id for CRM integration.
   - account_owner allows filtering entities based on ownership.
   - last_synced_time ensures data synchronization with CRM.
2. **Customer_Contact_relation Table**
   - Links customer_id to contact_id, establishing customer-specific relationships.
   - access_type defines access levels for contacts.
   - granted_at records access timestamps.
3. **customer_place_relation Table**
   - Maps customers to places, controlling visibility based on access_type.
   - granted_at records access timestamps.
4. **contacts Table**
   - Maintains contact details with Account_owner field linking contacts to specific owners.
5. **places Table**
   - Stores location-based details with Account_owner for ownership-based visibility.

Handling Customer-Specific Views

- Customers can be mapped to external CRM systems via the customer_mapping table.
- To retrieve all entities related to a specific account owner (e.g., Account Owner = “Bob"), filter using account_owner = 'Bob' in the relevant tables (customer_mapping, contacts, places).
  *SELECT * FROM contacts WHERE account_owner = 'Bob';*
- The customer_place_relation table governs access to places, and customer_contact_relation ensures visibility of relevant contacts.

Search Architecture

Oracle SQL Developer (OLTP)

1. Searching Contact Attributes (Title, Email, etc.)
   - Solution: Utilize Full-Text Indexing.
2. Searching for Entity Relationships
   - Solution: Use Foreign Key Indexing (place_id in contacts).
3. Searching Customer-Specific IDs (CRM Integrations)
   - Solution: Query the customer_mapping table.
4. Custom Filters (UI-Based Queries)
   - Solution: Implement Materialized Views to store precomputed query results.

Databricks Data Lake (OLAP)

1. Searching Contact Attributes
   - Solution: Use ZORDER Indexing.
2. Searching Entity Relationships
   - Solution: Optimize with Delta Lake Partitioning.
3. Searching Customer-Specific IDs
   - Solution: Implement Delta Live Tables (DLT) for real-time mapping.
4. Complex Filtering and Analytical Queries
   - Solution: Use SQL Warehouse for large-scale searches.

Performance Considerations and Tradeoff Discussions

Analysis of performance considerations and tradeoffs in Oracle SQL Developer (OLTP) and Databricks Data Lake (OLAP).

Key areas to optimize:

1. Indexing Strategy
2. Partitioning for Large Tables
3. Query Optimization & Execution Plans
4. Caching & Materialized Views
5. Scalability & Auto-scaling
6. Storage Efficiency

1. Indexing Strategy

Performance Consideration:

- **Goal:** Speed up lookups & joins while avoiding excessive index overhead.
- **Approach:** Use primary, foreign key, composite, and full-text indexes.

Tradeoff Discussion:

Tradeoff	**Pros**	**Cons**
More Indexes	Faster query execution	Slower insert/update operations
Fewer Indexes	Faster writes, less storage	Slower search queries, full table scans
Composite Indexes	Optimized for multi-column filters	Requires precise query design
Full-Text Indexes	Efficient text searches	Additional storage overhead

- **Oracle SQL Developer:** Use composite indexes only on frequently used multi-column queries.
- **Databricks:** Use ZORDER indexing to speed up searches while minimizing storage impact.

```
CREATE TABLE contacts
USING DELTA
PARTITIONED BY (place_id);
```

2. Partitioning for Large Tables

Performance Consideration:

- **Goal:** Avoid full table scans by reading only relevant data.
- **Approach:** Use partitioning by key fields (place_id, state_abbr).

Tradeoff Discussion:

Tradeoff	**Pros**	**Cons**
More Partitions	Faster queries (reads only relevant partitions)	Increased metadata and partition overhead
Fewer Partitions	Simpler management	More data scanned per query
Hash Partitioning	Evenly distributes data	Not useful for range-based filtering

- **Oracle SQL Developer:** Use LIST partitioning for places (by state) and HASH partitioning for contacts (by place_id).
- **Databricks:** Use Delta Lake partitioning and Auto-Compaction to manage performance.

```
IF query contains "state_abbr = 'NY'":
  SEARCH only within 'NY' partition
  IGNORE other partitions
RETURN results
```

4. Caching & Materialized Views

Performance Consideration:

- **Goal:** Reduce redundant computations by caching frequently accessed data.
- **Approach:** Use Materialized Views in Oracle and Delta Caching in Databricks.

Tradeoff Discussion:

Tradeoff	**Pros**	**Cons**
Using Materialized Views	Faster query performance	Needs regular refreshing
Using Delta Caching	Instant access for repeated queries	Consumes memory resources
Precomputed Views	Reduces CPU usage for heavy queries	Additional storage required

- **Oracle SQL Developer:** Use Materialized Views for customer-specific filtering.
- **Databricks:** Use Delta Caching for frequent UI queries.

5. Scalability & Auto-Scaling

Performance Consideration:

- **Goal:** Maintain performance as data grows (millions of records, concurrent queries).
- **Approach:** Use Elastic Scaling in Oracle and Auto-Scaling in Databricks.

Tradeoff Discussion:

Tradeoff	**Pros**	**Cons**
Auto-Scaling (Databricks)	Handles workload spikes	Higher cost during peak usage
Fixed-Sized Cluster (Databricks)	Predictable cost	May cause performance issues under load

- **Oracle SQL Developer:** Use Elastic Query Optimization.
- **Databricks:** Use Auto-Scaling for Large Query Loads.

6. Storage Efficiency

Performance Consideration:

- **Goal:** Minimize storage costs while maintaining performance.
- **Approach:** Use Data Compaction (Databricks) & Table Compression (Oracle).

```
OPTIMIZE contacts ZORDER BY (place_id, contact_id);
```

Tradeoff Discussion:

Tradeoff	**Pros**	**Cons**
Compacted Storage (Databricks)	Faster queries, less fragmentation	Higher upfront compute cost
Compressed Tables (Oracle)	Reduced disk space usage	Slightly slower insert performance

- **Oracle SQL Developer:** Use Table Compression for large tables.
- **Databricks:** Use Auto-Compaction in Delta Lake.

Future Improvements

As the system scales, improvements in performance, scalability, search efficiency, data governance, and AI-driven analytics are necessary.

1. Performance Optimization

- Query Caching: Implement Databricks Delta Caching and Materialized Views for frequently accessed data.
- Auto-Partitioning: Use Databricks Auto Optimize to dynamically partition data and improve query performance.

2. Scalability Enhancements

- Multi-Cloud Deployment: Leverage AWS (S3 + Databricks Delta Lake). Auto scaling of cluster for scalability

3. Advanced Search Enhancements and Updates

- AI-Driven Search Ranking: Prioritize relevant contacts using machine learning
- Real-Time Search Updates: Implement Change Data Capture (CDC)

4. Data Governance & Security

- PII Data Masking: Apply data masking to protect sensitive information like emails and phone numbers.

5. AI & Predictive Analytics

6. Future Proofing with Emerging Technologies

- Serverless Processing: Use Databricks Serverless SQL to reduce infrastructure costs during low usage periods.