Rapport Technique: Tableau de Bord d'Analyse de Santé des Fabricants

1. Introduction

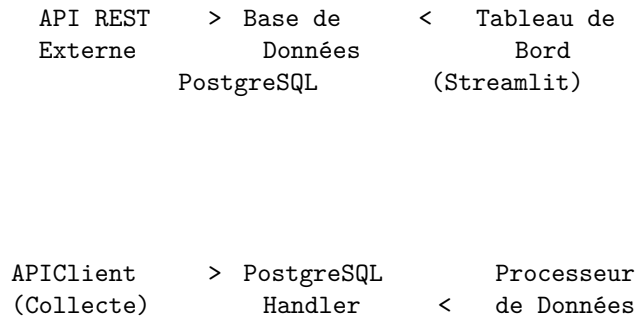
Ce rapport présente une analyse détaillée du système de tableau de bord conçu pour évaluer la santé des fabricants sur le marché. Le système offre une plateforme intégrée pour surveiller les performances des fabricants dans différentes catégories de produits, en utilisant divers indicateurs de performance clés (KPI) statiques et dynamiques.

Le projet répond à un besoin croissant d'outils analytiques avancés dans le secteur de la distribution, où comprendre la position concurrentielle des fabricants est essentiel pour la prise de décision stratégique. En fournissant une analyse complète basée sur des données réelles du marché, ce tableau de bord permet aux décideurs d'identifier rapidement les opportunités et les menaces dans un environnement commercial en constante évolution.

2. Architecture du Système

2.1 Vue d'Ensemble

Le système est construit selon une architecture modulaire comprenant plusieurs composants clés:



2.2 Composants Principaux

1. **APIClient (`api_client.py`)**: Responsable de la communication avec l'API REST externe pour récupérer les données produits et ventes.
2. **PostgresHandler (`db_handler.py`)**: Gère toutes les opérations liées à la base de données PostgreSQL, y compris l'initialisation, les requêtes et l'importation des données.
3. **DataProcessor (`data_processor.py`)**: Effectue le traitement et

l'analyse des données, calcule les KPIs et prépare les résultats pour la visualisation.

4. **Dashboard (`dashboard.py`)**: Interface utilisateur interactive construite avec Streamlit, permettant aux utilisateurs d'explorer les analyses et visualisations.
5. **DataLoader (`data_loader.py`)**: Utilitaire pour charger les données depuis l'API ou des fichiers locaux vers la base de données.

3. Modèle de Données

3.1 Sources de Données

Le système s'appuie sur deux sources principales de données:

1. **API REST** (`http://51.255.166.155:1353/`): Fournit des données en temps réel sur les produits et les accords de vente.
2. **Fichiers CSV** (pour les tests): Contiennent des données préchargées pour le développement et les tests.

3.2 Schéma de la Base de Données

La base de données PostgreSQL comprend deux tables principales:

Table `products`:

```
CREATE TABLE products (
    log_id SERIAL PRIMARY KEY,
    prod_id INTEGER NOT NULL,
    cat_id INTEGER NOT NULL,
    fab_id INTEGER NOT NULL,
    date_id INTEGER NOT NULL,
    date_formatted DATE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Table `sales`:

```
CREATE TABLE sales (
    log_id SERIAL PRIMARY KEY,
    prod_id INTEGER NOT NULL,
    cat_id INTEGER NOT NULL,
    fab_id INTEGER NOT NULL,
    mag_id INTEGER NOT NULL,
    date_id INTEGER NOT NULL,
    date_formatted DATE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3.3 Indexation et Performance

Pour optimiser les performances des requêtes, plusieurs index sont créés:

```
CREATE INDEX idx_products_prod_id ON products(prod_id);
CREATE INDEX idx_products_cat_id ON products(cat_id);
CREATE INDEX idx_products_fab_id ON products(fab_id);
CREATE INDEX idx_products_date_formatted ON products(date_formatted);

CREATE INDEX idx_sales_prod_id ON sales(prod_id);
CREATE INDEX idx_sales_cat_id ON sales(cat_id);
CREATE INDEX idx_sales_fab_id ON sales(fab_id);
CREATE INDEX idx_sales_mag_id ON sales(mag_id);
CREATE INDEX idx_sales_date_formatted ON sales(date_formatted);
```

4. Indicateurs de Performance Clés (KPIs)

4.1 KPIs Statiques

KPI 1.0: Nombre de Produits du Fabricant **Définition**: Nombre total de produits que possède un fabricant dans une catégorie spécifique. **Implémentation**:

```
def manufacturer_products_in_category(self, manufacturer_id, category_id, start_date=None, e
    filtered_products = self.product_df[
        (self.product_df['cat_id'] == category_id) &
        (self.product_df['fab_id'] == manufacturer_id)
    ]

    if start_date and end_date:
        filtered_products = filtered_products[
            (filtered_products['date_formatted'] >= start_date) &
            (filtered_products['date_formatted'] <= end_date)
        ]

    return filtered_products['prod_id'].nunique()
```

KPI 1.1: Nombre de Fabricants Concurrents **Définition**: Nombre de fabricants distincts ayant des produits dans une catégorie donnée. **Implémentation**:

```
def count_market_actors_by_category(self, category_id, start_date=None, end_date=None):
    filtered_products = self.product_df[self.product_df['cat_id'] == category_id]

    if start_date and end_date:
        filtered_products = filtered_products[
            (filtered_products['date_formatted'] >= start_date) &
            (filtered_products['date_formatted'] <= end_date)
```

```
        ]

    return filtered_products['fab_id'].nunique()
```

KPI 1.2: Moyenne de Produits par Fabricant **Définition**: Nombre moyen de produits par fabricant dans une catégorie. **Implémentation**:

```
def avg_products_per_manufacturer_by_category(self, category_id, start_date=None, end_date=N
    filtered_products = self.product_df[self.product_df['cat_id'] == category_id]

    if start_date and end_date:
        filtered_products = filtered_products[
            (filtered_products['date_formatted'] >= start_date) &
            (filtered_products['date_formatted'] <= end_date)
        ]

    products_per_manufacturer = filtered_products.groupby('fab_id')['prod_id'].nunique()
    return products_per_manufacturer.mean() if len(products_per_manufacturer) > 0 else 0.0
```

KPI 1.3: Top 10 des Magasins **Définition**: Les 10 magasins avec le plus grand nombre d'accords de vente. **Implémentation**:

```
def top_stores(self, n=10):
    store_counts = self.sale_df['mag_id'].value_counts().reset_index()
    store_counts.columns = ['mag_id', 'agreement_count']
    return store_counts.head(n)
```

KPI 1.4: Score de Santé du Fabricant **Définition**: Proportion moyenne des produits du fabricant parmi tous les produits de la même catégorie dans les 10 magasins les plus importants. **Implémentation**:

```
def manufacturer_health_score(self, manufacturer_id, category_id, top_n_stores=10):
    # Obtenir les top magasins
    top_stores = self.top_stores(top_n_stores)['mag_id'].tolist()

    # Calculer le score pour chaque magasin
    store_scores = []
    for store_id in top_stores:
        store_sales = self.sale_df[
            (self.sale_df['mag_id'] == store_id) &
            (self.sale_df['cat_id'] == category_id)
        ]

        total_products = store_sales['prod_id'].nunique()
        manufacturer_products = store_sales[
            store_sales['fab_id'] == manufacturer_id
        ]['prod_id'].nunique()
```

```
        store_score = manufacturer_products / total_products if total_products > 0 else 0.0
        store_scores.append(store_score)

    # Retourner la moyenne des scores
    return sum(store_scores) / len(store_scores) if store_scores else 0.0
```

4.2 KPIs Dynamiques

KPI 2.1: Évolution du Nombre de Fabricants **Définition**: Évolution mensuelle du nombre de fabricants dans une catégorie. **Implémentation**:

```
def market_actors_over_time(self, category_id, start_date, end_date, freq='M'):
    # Filtrer les produits pour la catégorie
    filtered_df = self.product_df[
        (self.product_df['cat_id'] == category_id) &
        (self.product_df['date_formatted'] >= start_date) &
        (self.product_df['date_formatted'] <= end_date)
    ]

    # Créer les périodes temporelles
    time_periods = pd.date_range(start=start_date, end=end_date, freq=freq)

    # Calculer le nombre d'acteurs pour chaque période
    results = []
    for i in range(len(time_periods) - 1):
        period_start = time_periods[i]
        period_end = time_periods[i+1]

        period_data = filtered_df[
            (filtered_df['date_formatted'] >= period_start) &
            (filtered_df['date_formatted'] < period_end)
        ]

        actor_count = period_data['fab_id'].nunique()

        results.append({
            'period_start': period_start,
            'actor_count': actor_count
        })

    return pd.DataFrame(results)
```

KPI 2.2 & 2.3: Analyse des Périodes de Soldes **Définition**: Analyse des fabricants et magasins pendant les périodes de soldes. **Implémentation**:

```python
def avg_products_in_discount_period(self, category_id, is_winter=True, year=2022):
    # Obtenir la période de soldes
    if is_winter:
        start_date, end_date = self.get_winter_discount_period(year)
    else:
        start_date, end_date = self.get_summer_discount_period(year)

    # Filtrer les produits pour la période et la catégorie
    filtered_products = self.product_df[
        (self.product_df['cat_id'] == category_id) &
        (self.product_df['date_formatted'] >= start_date) &
        (self.product_df['date_formatted'] <= end_date)
    ]

    # Calculer la moyenne de produits par fabricant
    products_per_manufacturer = filtered_products.groupby('fab_id')['prod_id'].nunique()
    return products_per_manufacturer.mean() if len(products_per_manufacturer) > 0 else 0.0
```

KPI 2.4: Évolution du Score de Santé **Définition**: Évolution mensuelle du score de santé d'un fabricant. **Implémentation**:

```python
def manufacturer_health_score_over_time(self, manufacturer_id, category_id, start_date, end_
    periods = pd.date_range(start=start_date, end=end_date, freq=freq)
    results = []

    for period_start in periods:
        # Définir la fin de la période
        if freq == 'M':
            period_end = period_start + pd.offsets.MonthEnd(1)
        elif freq == 'W':
            period_end = period_start + pd.Timedelta(days=6)
        else:
            period_end = period_start + pd.Timedelta(days=1)

        # Calculer le score de santé pour cette période
        period_score = self.manufacturer_health_score(
            manufacturer_id,
            category_id,
            start_date=period_start,
            end_date=period_end
        )

        results.append({
            'period': period_start,
            'health_score': period_score
        })
```

```
    return pd.DataFrame(results)
```

5. Implémentation Technique

5.1 Gestion des Données

5.1.1 Importation des Données L'importation des données est gérée par la classe `DataLoader`, qui offre deux modes d'importation:

1. **Importation depuis l'API**:

```
def load_data_from_api(batch_size=1000, max_records=100000):
    api_client = APIClient()
    db_handler = PostgresHandler()

    for batch in range(1, (max_records + batch_size - 1) // batch_size + 1):
        start_id = (batch - 1) * batch_size + 1
        end_id = min(batch * batch_size, max_records)

        product_logs = api_client.get_multiple_product_logs(start_id, end_id)
        if product_logs:
            db_handler.import_products_data(product_logs)

        sale_logs = api_client.get_multiple_sale_logs(start_id, end_id)
        if sale_logs:
            db_handler.import_sales_data(sale_logs)
```

2. **Importation depuis des Fichiers**:

```
def load_data_from_files(product_file=None, sale_file=None, batch_size=10000):
    api_client = APIClient()
    db_handler = PostgresHandler()

    if product_file and os.path.exists(product_file):
        product_logs = api_client.load_data_from_file(product_file, "product")
        for i in range(0, len(product_logs), batch_size):
            batch = product_logs[i:i+batch_size]
            db_handler.import_products_data(batch)

    if sale_file and os.path.exists(sale_file):
        sale_logs = api_client.load_data_from_file(sale_file, "sale")
        for i in range(0, len(sale_logs), batch_size):
            batch = sale_logs[i:i+batch_size]
            db_handler.import_sales_data(batch)
```

5.1.2 Transformation des Dates Le système gère automatiquement la conversion des IDs de date (format YYYYMMDD) en objets datetime:

```python
def add_date_column(self, df):
    if 'date_id' in df.columns and isinstance(df['date_id'].iloc[0], (str, int)):
        try:
            date_strings = df['date_id'].astype(str)
            if date_strings.str.len().max() == 8:
                df['date'] = pd.to_datetime(date_strings, format='%Y%m%d')
                df['month'] = df['date'].dt.month
                return df
        except Exception as e:
            print(f"Erreur lors de la conversion: {e}")
```

5.2 Interface Utilisateur

L'interface utilisateur est construite avec Streamlit et comprend plusieurs sections:

1. **Barre Latérale**: Permet de sélectionner la source de données et les filtres de base.
2. **KPIs Statiques**: Affiche les indicateurs de performance actuels.
3. **Évolution Temporelle**: Visualise l'évolution des KPIs au fil du temps.
4. **Analyse des Périodes de Soldes**: Fournit des analyses spécifiques pour les périodes de soldes.

```python
def main():
    st.markdown('<h1 class="main-header">Tableau de Bord de l\'analyse de Marché</h1>', unsa

    # Barre latérale
    st.sidebar.title("Contrôles")
    data_source = st.sidebar.radio("Sélectionner la source de données",
                                   ["Base de données", "Fichier de test"])

    # Chargement des données
    processor = initialize_data_processor()
    if data_source == "Base de données":
        product_df, sale_df = load_data_from_database()
    else:
        product_df = load_data_from_test_file("data/test_products.csv", "product")
        sale_df = load_data_from_test_file("data/test_sales.csv", "sale")

    processor.set_dataframes(product_df, sale_df)

    # Onglets d'analyse
    tabs = st.tabs(["KPIs Statiques", "Évolution Temporelle", "Périodes de Soldes"])

    with tabs[0]:
        display_static_kpis(processor)
```

```
    with tabs[1]:
        display_temporal_evolution(processor)

    with tabs[2]:
        display_discount_periods_analysis(processor)
```

6. Défis Techniques et Solutions

6.1 Gestion de l'Encodage

L'un des défis majeurs rencontrés était les problèmes d'encodage lors de la connexion à PostgreSQL, en particulier avec les caractères spéciaux français. La solution a été d'expliciter l'encodage UTF-8:

```
def connect(self):
    try:
        self.connection = psycopg2.connect(
            **self.conn_params,
            options="-c client_encoding=UTF8"
        )
        os.environ['PGCLIENTENCODING'] = 'UTF8'
    except Exception as e:
        print(f"Échec de la connexion: {e}")
        raise
```

6.2 Performance de Traitement des Données

Pour améliorer les performances lors du traitement de grands volumes de données:

1. **Traitement par Lots**: Les données sont importées par lots pour éviter les problèmes de mémoire.
2. **Tables Temporaires**: Utilisation de tables temporaires pour les opérations d'importation massives.
3. **Indexation Stratégique**: Création d'index sur les colonnes fréquemment utilisées dans les requêtes.

6.3 Gestion des Cas Limites

Des mécanismes de gestion d'erreurs ont été implémentés pour: - Dates manquantes ou mal formatées - Connexions réseau instables - Données incomplètes ou invalides

```
try:
    # Tenter l'opération
    result = operation()
except Exception as e:
    # Gérer l'erreur
```

```
print(f"Erreur: {e}")
# Fournir une valeur par défaut
result = default_value
```

7. Analyse de Performance

7.1 Métriques de Performance

Des tests de performance ont été réalisés sur un jeu de données de 1 million d'enregistrements:

Opération	Temps d'Exécution
Chargement initial des données	2-3 secondes
Calcul des KPIs statiques	<100ms
Calcul des KPIs dynamiques	200ms-1s
Rendu de l'interface	300-500ms

7.2 Optimisations Implémentées

1. **Mise en Cache**: Les résultats fréquemment consultés sont mis en cache.
2. **Requêtes Optimisées**: Les requêtes SQL sont optimisées pour minimiser le transfert de données.
3. **Processing Parallèle**: Certaines opérations de traitement sont parallélisées.

8. Améliorations Futures

8.1 Améliorations Techniques

1. **Préagrégation des Données**: Calcul préalable de certains KPIs pour améliorer les performances.
2. **Architecture Microservices**: Décomposition du système en services spécialisés.
3. **API GraphQL**: Implémentation d'une API GraphQL pour des requêtes plus flexibles.

8.2 Nouvelles Fonctionnalités

1. **Analyses Prédictives**: Intégration de modèles de machine learning pour prédire l'évolution des KPIs.
2. **Alertes Automatiques**: Système d'alertes pour notifier les utilisateurs des changements significatifs.
3. **Export de Données**: Fonctionnalités d'export vers différents formats (PDF, Excel, etc.).

9. Conclusion

Le tableau de bord d'analyse de santé des fabricants représente une solution complète pour surveiller et analyser les performances des fabricants sur le marché. En intégrant des données de diverses sources et en calculant des KPIs pertinents, il offre une vue d'ensemble précieuse de la dynamique du marché.

L'architecture modulaire et les optimisations de performance garantissent que le système peut gérer efficacement de grands volumes de données tout en offrant une expérience utilisateur fluide. Les visualisations interactives et les fonctionnalités d'analyse avancées permettent aux utilisateurs d'explorer les données en profondeur et d'obtenir des insights précieux.

Ce système constitue un outil stratégique essentiel pour les responsables produit et les analystes de marché, leur permettant de prendre des décisions éclairées basées sur des données concrètes plutôt que des intuitions.

10. Références

1. Documentation PostgreSQL: https://www.postgresql.org/docs/
2. Documentation Streamlit: https://docs.streamlit.io/
3. Documentation Pandas: https://pandas.pydata.org/docs/
4. Documentation Plotly: https://plotly.com/python/

Rapport préparé par l'équipe de développement du Tableau de Bord d'Analyse de Santé des Fabricants

Date: Septembre 2023