Projet de l’équipe D pour le cours de Megadonnées à l’UPEC : Flight Catcher

Groupe D

Massil Kouchit  
 Master 2 – Logiciel Sûr   
massil.kouchit@etu.u-pec.fr

Yanis Ben Aied  
Master 2 – Logiciel Sûr  
yanis.ben-aied@etu.u-pec.frRayane Slimani  
Master 2 – Logiciel Sûr  
aissa-rayane.slimani@etu.u-pec.fr

Rayan Dansou  
Master 2 – Logiciel Sûr  
yao-rayan.dansou@etu.u-pec.frAito Meignan  
Master 2 – Logiciel Sûr  
aito-florentin.meignan@etu.u-pec.fr

# **Resumé - Flight Catcher est un projet d’analyse des données aériennes commerciales visant à évaluer les retards, optimiser les performances et générer des intuition pour les compagnies, les aéroports, et les passagers. Il s’appuie sur SPARK, FASTAPI, et un analyseur syntaxique SQL pour des requêtes personnalisées.**

Keywords— Flight Catcher, SPARK, FASTAPI, DASH PLOTLY, Parser, SQL, Parquet, DuckDB

# Introduction

## Contexte et enjeux du transport aérien moderne

Dans un monde de plus en plus interconnecté, le transport aérien occupe une place centrale, permettant de relier les continents et de faciliter les échanges économiques, culturels et humains. Cependant, cette importance s'accompagne de défis significatifs pour les compagnies aériennes et les gestionnaires d’aéroports, notamment la gestion des retards de vols, qui représente une source majeure de perturbations pour les passagers et les opérations. Ces retards, souvent causés par des facteurs tels que les conditions météorologiques, les contraintes de gestion du trafic aérien ou des problèmes techniques, engendrent des coûts économiques et sociaux conséquents.

Face à ces enjeux, le développement de solutions basées sur l’analyse des données aéronautiques devient crucial. C’est dans ce contexte que s’inscrit le projet **Flight Catcher**, qui exploite le jeu de données **Flight Status Prediction[1]** de Kaggle. L’objectif est de concevoir une interface utilisateur intuitive et performante, capable de traiter et de visualiser des données complexes afin d’aider les acteurs du secteur à anticiper et réduire les retards. Les passagers des vols pourront également utiliser l’interface afin de choisir le vol qui leur correspond le plus. Ce projet ambitionne de transformer des données brutes en informations exploitables pour optimiser la prise de décision et améliorer l'expérience utilisateur dans le domaine du transport aérien.

## Structure du document

Le présent rapport s’articule autour de plusieurs sections détaillant chaque étape clé de notre projet. La **section II** introduit les choix techniques initiaux, avec une présentation de l'architecture partie serveur basée sur FASTAPI et DuckDB, ainsi que les premières analyses réalisées sur le jeu de données. La **section III** explore le remplacement de DuckDB par Apache Spark, en mettant en lumière les avantages de cette solution pour des analyses à plus grande échelle et des applications de machine learning. La **section IV** se concentre sur le développement de l’interface utilisateur, où l’utilisation de Dash Plotly a permis de concevoir une interface utilisateur interactive et intuitive pour visualiser les données. La **section V** évalue les performances de notre système, notamment en termes de rapidité et de scalabilité, tandis que la **section VI** propose des scénarios d'application concrets illustrant l'utilité de notre interface dans divers contextes. Enfin, nous concluons avec une synthèse des résultats obtenus et des perspectives d’amélioration pour ce projet.

## Utilisateurs potentiels

L’interface développée dans le cadre du projet **Flight Catcher** a été conçue pour répondre aux besoins d’une large variété d’utilisateurs, qu’ils soient passagers occasionnels ou professionnels du secteur aérien.

D’une part, les **passagers individuels** constituent un groupe clé d’utilisateurs. Grâce à cette interface, ils pourront accéder à des informations claires et précises sur les vols, comme les prévisions de retards ou les tendances générales pour une destination donnée. Ces données leur permettront de mieux planifier leurs déplacements et de choisir les options les plus fiables pour leurs voyages, optimisant ainsi leur expérience.

D’autre part, les **professionnels du secteur aérien** bénéficieront d’outils puissants pour améliorer leurs opérations. Les **aéroports** pourraient utiliser cette interface pour optimiser la gestion des ressources au sol, telles que les pistes ou les portes d’embarquement, en anticipant les retards. Les **compagnies aériennes**, quant à elles, pourront se servir de ces analyses pour ajuster leurs plannings, réduire les impacts des retards sur les connexions, et améliorer la satisfaction de leurs clients. En outre, les **organismes de régulation** ou les prestataires de services liés au transport aérien (comme les gestionnaires de bagages ou les équipes de maintenance) pourraient exploiter ces données pour coordonner leurs activités et réduire les inefficacités opérationnelles.

Enfin, cette interface s’adresse également à l’ensemble des **acteurs de l’écosystème aéronautique**, tels que les analystes de données ou les décideurs stratégiques, qui peuvent s’appuyer sur les informations fournies pour identifier des tendances, évaluer les performances et prendre des décisions éclairées visant à améliorer la fiabilité et l’efficacité globale du transport aérien.

En rendant les données accessibles et exploitables pour ces divers profils d’utilisateurs, **Flight Catcher** se positionne comme une solution au service de la fluidité et de l’efficacité du transport aérien.

Outre les passagers et les professionnels directs du secteur aérien, l'interface **Flight Catcher** pourrait également intéresser les **investisseurs** et actionnaires cherchant à optimiser leurs choix d'investissement dans l'industrie aérienne. Ces acteurs financiers, qu’ils soient intéressés par les **compagnies aériennes**, les **modèles d’avions**, ou même les **aéroports**, pourraient utiliser les données fournies par l'interface pour identifier les acteurs les plus performants en termes de ponctualité et de gestion des retards. En analysant les tendances de performance, ils pourront prendre des décisions éclairées sur les investissements les plus prometteurs, en privilégiant les compagnies aériennes qui démontrent une gestion optimale des flux et une réduction significative des perturbations. Cela pourrait aussi les aider à repérer des zones géographiques ou des infrastructures (aéroports, compagnies ou modèles d'avions) particulièrement efficaces, réduisant ainsi les risques financiers et maximisant les retours sur investissement.

Au-delà des acteurs traditionnels du secteur aérien, plusieurs autres profils pourraient être intéressés par l'interface **Flight Catcher**. Par exemple, les **fournisseurs de services technologiques** (tels que les entreprises spécialisées dans les systèmes de gestion du trafic aérien ou les technologies de suivi de vol) pourraient exploiter ces données pour enrichir leurs propres solutions et proposer des services plus efficaces. De même, les **consultants en gestion des risques** pourraient se servir des analyses pour aider leurs clients à minimiser les impacts des retards et à optimiser la planification des ressources.

Les **voyagistes et agences de voyages** pourraient également trouver une utilité directe dans cette interface, leur permettant de mieux conseiller leurs clients en fonction des performances des compagnies aériennes et des aéroports, ou encore d'offrir des services personnalisés, comme des recommandations sur les horaires de vol les plus fiables.

Enfin, les **chercheurs et universitaires** travaillant dans des domaines tels que l’ingénierie du transport, les sciences des données, ou l’économie des transports pourraient tirer parti de l'interface pour mener des études sur la ponctualité des vols, l’impact des retards sur les performances économiques, ou encore l’évolution des technologies aéronautiques.

L’interface **Flight Catcher**, par son approche analytique et ses fonctionnalités adaptées à une diversité d’utilisateurs, pourrait donc se révéler un outil précieux pour une large gamme d'acteurs investis dans l’amélioration de l'efficacité et de la rentabilité du secteur aérien.

## Le jeu de donnée initial

Le jeu de données utilisé dans le projet **Flight Catcher** provient de la plateforme Kaggle, un site de référence pour les compétitions et analyses de données. Ce jeu, intitulé **Flight Delay Dataset 2018-2022**, a été créé pour permettre l'étude des retards et des performances des vols commerciaux aux États-Unis sur une période de cinq ans, de 2018 à 2022. Il contient des informations détaillées telles que les retards, les compagnies aériennes, les aéroports d’origine et de destination, les horaires de vol, le temps de trajet, ainsi que les distances parcourues.

Avec un volume total de 2 Go, le fichier est fourni au format Parquet, un format bien adapté aux grandes volumétries de données. Ce choix offre plusieurs avantages : une structure orientée colonnes qui optimise les analyses par lots et des capacités avancées de compression, ce qui réduit considérablement la taille des fichiers tout en maintenant des temps d'accès rapides. Ces caractéristiques font du format Parquet une solution idéale pour le traitement efficace des données volumineuses, limitant ainsi les ressources nécessaires tout en conservant une grande flexibilité pour les analyses.

Ce jeu de données sera exploré en détail dans une section ultérieure du rapport, où nous expliquerons comment ses différentes caractéristiques ont guidé nos choix techniques et analytiques, ainsi que les premières observations tirées des données.

# Description des colonnes du dataset

Voici une explication des colonnes de notre dataset :

## Informations générales sur le vol

* **Year** : Année (yyyy).
* **Quarter** : Trimestre (1-4).
* **Month** : Mois (1-12).
* **DayofMonth** : Jour du mois (1-31).
* **DayOfWeek** : Jour de la semaine (1-7).
* **FlightDate** : Date du vol (format : aaaa-mm-jj).

## Informations sur la compagnie aérienne

* **Airline** : Compagnie aérienne (code unique).
* **Marketing\_Airline\_Network** : Code unique de la compagnie marketing. Lorsqu'un même code est utilisé par plusieurs compagnies, un suffixe numérique est ajouté pour les utilisateurs précédents (ex. : PA, PA(1), PA(2)). Utilisez ce champ pour les analyses sur plusieurs années.
* **Operated\_or\_Branded\_Code\_Share\_Partners** : Partenaires opérateurs ou de partage de code.
* **DOT\_ID\_Marketing\_Airline** : Numéro d'identification attribué par le Département des Transports des États-Unis (DOT) pour identifier une compagnie aérienne unique.
* **IATA\_Code\_Marketing\_Airline** : Code attribué par l'IATA(International Air Transport Association), souvent utilisé pour identifier une compagnie aérienne. Ce code n'est pas toujours unique.
* **Flight\_Number\_Marketing\_Airline** : Numéro de vol.
* **Operating\_Airline** : Code unique de la compagnie aérienne opératrice.
* **DOT\_ID\_Operating\_Airline** : Numéro d'identification attribué par le DOT pour la compagnie aérienne opératrice.
* **IATA\_Code\_Operating\_Airline** : Code IATA de la compagnie aérienne opératrice.
* **Tail\_Number** : Numéro de queue de l'avion.
* **Flight\_Number\_Operating\_Airline** : Numéro de vol de la compagnie opératrice.

## Informations sur l'origine et la destination

* **Origin** : Code de l'aéroport d'origine.
* **OriginAirportID** : Identifiant de l'aéroport d'origine attribué par le DOT. Utilisez ce champ pour les analyses sur plusieurs années.
* **OriginAirportSeqID** : Identifiant séquentiel de l'aéroport d'origine.
* **OriginCityMarketID** : Identifiant du marché de la ville d'origine.
* **OriginCityName** : Nom de la ville de l'aéroport d'origine.
* **OriginState** : Code de l'État de l'aéroport d'origine.
* **OriginStateFips** : Code FIPS (Federal Information Processing Standards) de l'État de l'aéroport d'origine.
* **OriginStateName** : Nom de l'État de l'aéroport d'origine.
* **OriginWac** : Code de la zone mondiale de l'aéroport d'origine.
* **Dest** : Code de l'aéroport de destination.
* **DestAirportID** : Identifiant de l'aéroport de destination attribué par le DOT.
* **DestAirportSeqID** : Identifiant séquentiel de l'aéroport de destination.
* **DestCityMarketID** : Identifiant du marché de la ville de destination.
* **DestCityName** : Nom de la ville de l'aéroport de destination.
* **DestState** : Code de l'État de l'aéroport de destination.
* **DestStateFips** : Code FIPS de l'État de l'aéroport de destination.
* **DestStateName** : Nom de l'État de l'aéroport de destination.
* **DestWac** : Code de la zone mondiale de l'aéroport de destination.

## Informations sur les horaires

* **CRSDepTime** : Heure de départ prévue (heure locale : hhmm).
* **DepTime** : Heure de départ réelle (heure locale : hhmm).
* **DepDelay** : Différence en minutes entre l'heure prévue et l'heure réelle de départ (valeurs négatives pour des départs anticipés).
* **DepDelayMinutes** : Délai de départ en minutes (valeurs anticipées réglées sur 0).
* **DepDel15** : Indicateur de délai de départ de 15 minutes ou plus (1=Oui).
* **DepartureDelayGroups** : Intervalles de délai de départ, par tranche de 15 minutes.
* **DepTimeBlk** : Intervalle horaire de départ prévu (bloc CRS).
* **TaxiOut** : Temps de roulage au départ, en minutes.
* **WheelsOff** : Heure de décollage (heure locale : hhmm).
* **WheelsOn** : Heure d'atterrissage (heure locale : hhmm).
* **TaxiIn** : Temps de roulage à l'arrivée, en minutes.
* **CRSArrTime** : Heure d'arrivée prévue (heure locale : hhmm).
* **ArrTime** : Heure d'arrivée réelle (heure locale : hhmm).
* **ArrDelay** : Différence en minutes entre l'heure prévue et l'heure réelle d'arrivée (valeurs négatives pour des arrivées anticipées).
* **ArrDelayMinutes** : Délai d'arrivée en minutes (valeurs anticipées réglées sur 0).
* **ArrDel15** : Indicateur de délai d'arrivée de 15 minutes ou plus (1=Oui).
* **ArrivalDelayGroups** : Intervalles de délai d'arrivée, par tranche de 15 minutes.
* **ArrTimeBlk** : Intervalle horaire d'arrivée prévue (bloc CRS).

## Informations sur l'annulation et les détournements

* **Cancelled** : Indicateur de vol annulé (1=Oui).
* **Diverted** : Indicateur de vol détourné (1=Oui).
* **DivAirportLandings** : Nombre d'atterrissages dans des aéroports de détournement.

## Informations sur la durée et la distance du vol

* **CRSElapsedTime** : Durée prévue du vol, en minutes.
* **ActualElapsedTime** : Durée réelle du vol, en minutes.
* **AirTime** : Temps de vol effectif, en minutes.
* **Distance** : Distance entre les aéroports (en miles).
* **DistanceGroup** : Intervalles de distance, par tranche de 250 miles.

## Exemples d'enregistrements

FlightDate : 2018-01-23

Airline : Endeavor Air Inc.

Cancelled : False

Diverted : False

ArrTime : 1256.0

AirTime : 38.0

# Mise en place de la partie serveur "minimal" et premières requêtes

## Choix techniques

Afin de développer l’interface d’analyse des données, nous avons opté pour l’utilisation de **FASTAPI** en tant que framework pour le développement de l’API côté serveur. Ce choix repose sur plusieurs avantages clés de FASTAPI : sa légèreté, ses performances élevées, et sa capacité à générer automatiquement une documentation des points d’accès (endpoints) via Swagger UI, ce qui facilite le développement collaboratif. Par rapport à des alternatives comme **Flask** ou **Django**, FASTAPI se distingue par sa prise en charge native d’asyncio, qui optimise la gestion des requêtes simultanées. Cette caractéristique est particulièrement utile pour notre projet, où des requêtes fréquentes et intensives sur les données sont attendues. Cependant, si Django offre un écosystème riche et intégré pour des projets nécessitant des fonctionnalités web complexes, il aurait pu être perçu comme trop lourd pour notre besoin spécifique, centré principalement sur des appels API performants.

Pour la partie base de données, nous avons choisi **DuckDB** comme moteur analytique principal. DuckDB s’est révélé particulièrement adapté au traitement de fichiers volumineux, notamment au format **Parquet**, grâce à son intégration native qui permet d’effectuer des requêtes SQL directement sur ces fichiers. Cela évite d’avoir à importer les données dans une base intermédiaire, réduisant à la fois le temps de prétraitement et l’utilisation de ressources mémoire. Par exemple, des benchmarks indiquent que DuckDB consomme en moyenne 40 à 70 % moins de mémoire que des bases relationnelles classiques comme **PostgreSQL** ou **MySQL** pour des opérations similaires sur des fichiers de grande taille. Comparé à des solutions distribuées comme **Apache Spark**, DuckDB est limité par son exécution locale mais se montre beaucoup plus rapide et efficient pour des volumes de données jusqu’à plusieurs dizaines de gigaoctets, ce qui correspond à notre cas.

En revanche, une limite notable de DuckDB réside dans sa scalabilité : il n’est pas conçu pour traiter des charges massives sur des clusters distribués. Si nos besoins venaient à évoluer vers des analyses nécessitant une parallélisation étendue ou une intégration avec des flux de données en temps réel, Spark pourrait être une alternative plus appropriée, malgré son coût en ressources et sa complexité d’installation.

Ainsi, le choix initial de **FASTAPI** et **DuckDB**, optimisé pour la légèreté et l’efficacité mémoire, reflétait un compromis judicieux entre performance, simplicité et adaptation à l’échelle de notre projet. Cependant, face à des besoins croissants en termes de scalabilité et d’analyse distribuée, notamment pour des volumes de données dépassant les capacités locales, nous avons dû basculer sur **Apache Spark**. Spark permet une gestion plus robuste des données massives grâce à sa capacité à exécuter des calculs sur des clusters distribués, offrant ainsi une solution adaptée à des traitements plus intensifs et complexes.

## Chargement des données

Les données du jeu Flight Status Prediction sont stockées au format Parquet, ce qui offre des avantages significatifs en termes de compression et de performance de lecture. Ce format a été choisi en raison de sa capacité à réduire la taille des fichiers tout en maintenant un accès rapide aux données, ce qui est essentiel pour traiter les 2 Go d’informations contenues dans ce jeu de données.

Grâce à DuckDB, le fichier Parquet peut être chargé directement dans une table temporaire et interagit avec des requêtes SQL classiques. Cette méthode supprime le besoin de transformations complexes et permet de tirer parti de la simplicité des requêtes SQL pour récupérer les informations pertinentes.

## Premiers objectifs d'analyse

Une fois les données accessibles via DuckDB, nous avons défini plusieurs types de requêtes pour répondre aux principaux besoins analytiques. Ces requêtes constituent les fondations de notre partie serveur et permettent d’extraire des statistiques utiles sur les vols commerciaux. Voici les premiers objectifs couverts par ces analyses (Exemples) :

Identification des compagnies aériennes : L’objectif est de récupérer la liste des compagnies présentes dans le jeu de données afin d’avoir un aperçu des acteurs impliqués.

Analyse des annulations : Le calcul du pourcentage de vols annulés a été réalisé selon deux approches :

Par année pour observer l’évolution des annulations au fil du temps.

Depuis une date spécifique pour analyser les tendances plus récentes.

Analyse des retards : De manière similaire aux annulations, l’étude des vols retardés a été menée par année et depuis une date donnée. Ces statistiques permettent d’identifier les périodes critiques où les retards sont les plus fréquents.

Classement des compagnies aériennes : Deux classements ont été mis en place :

Les 10 compagnies aériennes les plus utilisées, en fonction du nombre total de vols enregistrés.

Les 10 compagnies aériennes les plus performantes, en tenant compte à la fois du taux d’annulation et des délais moyens de départ.

Performance par État : Une analyse a été réalisée pour classer les États en fonction de leurs performances, notamment en évaluant le taux d’annulations et les retards moyens des vols.

Analyse des vols détournés : Enfin, le pourcentage de vols détournés a été étudié pour fournir une vision complète des perturbations.

Les requêtes définies avec DuckDB ont été intégrées dans la partie serveur via FASTAPI. Chaque requête est exposée sous forme d’endpoint API, permettant ainsi à l’interface utilisateur de récupérer les données analytiques en temps réel. Cette architecture garantit une communication fluide entre la partie serveur et l’interface utilisateur, tout en tirant parti des performances offertes par DuckDB.

Une image contenant texte, capture d’écran, conception

Le contenu généré par l’IA peut être incorrect.

# Remplacement de DuckDB par Spark

À la suite des premières phases d’analyse, certaines limites de DuckDB ont été identifiées, notamment pour le Machine Learning. Pour surmonter ces limites et répondre aux besoins en matière de scalabilité et d’efficacité, le choix a été fait de remplacer DuckDB par Apache Spark.

## Apache Spark

Apache Spark est un framework de traitement de données distribué qui offre des performances exceptionnelles pour les analyses de grandes volumétries. Contrairement à DuckDB, qui opère principalement en local, Spark permet de distribuer les calculs sur plusieurs nœuds, rendant le traitement plus rapide et plus efficace.

De plus, Spark s’avère particulièrement adapté si l’on souhaite intégrer des traitements de Machine Learning dans le futur. Grâce à sa bibliothèque MLlib, Spark propose des outils performants pour l’entraînement et l’exécution de modèles prédictifs sur des données massives.

L’un des principaux avantages de Spark réside dans son support natif des fichiers au format Parquet, tout comme DuckDB. Cette compatibilité a facilité la migration des requêtes existantes vers Spark sans nécessiter de transformations des données. De plus, Spark dispose d’une API SQL très performante, qui nous a permis de conserver une approche basée sur des requêtes structurées tout en bénéficiant de temps de réponse améliorés sur les analyses complexes.

## Nouvelles analyses réalisées avec Spark

Avec Spark, nous avons pu enrichir notre ensemble de requêtes analytiques en introduisant de nouvelles analyses plus avancées. Parmi celles-ci, on retrouve :

L’analyse des retards par heure de départ : Cette étude permet de déterminer les heures où les retards sont les plus fréquents, offrant ainsi une meilleure compréhension des périodes critiques pour les vols.

Le classement des aéroports en fonction de leur performance :

Cette analyse classe les aéroports en tenant compte de leur taux de retards et d’annulations. Elle permet d’identifier les infrastructures les plus performantes et celles qui nécessitent des améliorations.

L’étude de la corrélation entre les retards au départ et à l’arrivée : Cette analyse met en lumière les relations entre les retards enregistrés au départ des vols et les retards constatés à l’arrivée, afin de mieux comprendre les facteurs d’impact.

La distribution des vols par distance parcourue : Cette analyse segmente les vols en plusieurs catégories (courts, moyens et longs trajets) afin d’observer la répartition des distances dans le jeu de données.

La prédiction des retards futurs : Grâce aux capacités d’apprentissage automatique intégrées à Spark, nous avons posé les bases d’un modèle prédictif qui prend en compte des variables comme l’heure de départ, l’origine, la destination et la distance parcourue pour anticiper les retards.

Identify applicable funding agency here. If none, delete this text box.

## Adaptation de la partie serveur avec Spark

L’intégration d’Apache Spark dans la partie serveur développé avec FASTAPI a nécessité plusieurs ajustements. Une configuration spécifique de Spark a été mise en place pour assurer une lecture optimale des fichiers Parquet et permettre l’exécution rapide des nouvelles requêtes. Chaque analyse a ensuite été exposée sous forme d’endpoint API, afin de garantir une communication fluide entre la partie serveur et l’interface utilisateur.

# Développement de l’interface utilisateur avec Dash Plotly

Pour le développement de l’interface utilisateur, nous avons sélectionné le framework Dash Plotly, qui constitue une surcouche de React.js pour Python. Dash facilite la création d’applications web interactives pour la visualisation de données, tout en permettant d’intégrer des librairies analytiques comme Pandas pour le traitement des données.

L’interface utilisateur a été conçue pour interagir directement avec la partie serveur FASTAPI, offrant ainsi un tableau de bord interactif capable d’afficher les résultats des requêtes sous forme de tableaux ou de graphiques.

L’application repose sur une structure simple où chaque bouton permet de lancer une requête API spécifique et d’afficher les résultats dans un composant graphique ou tabulaire. Par exemple :

Affichage de la liste des compagnies aériennes : Lorsqu’un utilisateur clique sur le bouton correspondant, l’application récupère les données via un endpoint API et génère un tableau interactif grâce aux composants de Dash.

Calendrier des vols annulés : Une visualisation sous forme de carte thermique (heatmap) est générée pour représenter le pourcentage de vols annulés par mois et par année.

Enfin, Dash Bootstrap Components a été intégré pour assurer un design soigné et une expérience utilisateur optimale grâce à des styles modernes basés sur Bootstrap.

Une image contenant texte, Police, capture d’écran

Le contenu généré par l’IA peut être incorrect.

# Analyseur Syntaxique

## Intégration d’un Parser SQL dans Flight Catcher

Dans le cadre du projet Flight Catcher, un parser SQL a été intégré afin de permettre un traitement plus fluide et diversifié des requêtes sur les données. Ce composant joue un rôle clé en transformant des requêtes en langage naturel en instructions SQL précises et optimisées, adaptées à la structure des données du projet.

## Identification des Requêtes

Une des premières étapes du processus consiste à classer les requêtes. Le parser reconnaît différents types d’interrogations, qu’il s’agisse de calculs d’agrégats, de comparaisons ou encore de recherches temporelles. Cela permet de guider le traitement vers les modules spécialisés pour chaque type de requête.

## Extraction et Traduction des Conditions

Les conditions exprimées dans les requêtes textuelles, comme des comparaisons ou des critères temporels, sont extraites automatiquement. Par exemple, une plage de dates ou des correspondances spécifiques, telles que "vols de New York à Los Angeles" sont détectées et traduites en clauses SQL cohérentes avec la structure de la base de données.

## Mapping Automatique des Colonnes

Pour faciliter l’écriture des requêtes, le parser peut associer des termes génériques à des noms de colonnes spécifiques. Par exemple, un terme comme "airline" est automatiquement relié à la colonne pertinente dans la base de données. Cela permet une plus grande souplesse d’utilisation et réduit les risques d’erreur.

## Optimisation des Performances

Le parser ne se contente pas de traduire les requêtes : il optimise également leur exécution. Grâce à des techniques telles que la mise en cache des résultats ou la pré-compilation des motifs d’analyse, les requêtes fréquentes sont traitées plus rapidement. Ces optimisations garantissent une utilisation efficace des ressources, même pour des volumes de données importants.

## Exemple d’Utilisation

Prenons l’exemple d’une requête textuelle demandant "le délai moyen au départ des vols de New York à Los Angeles en janvier 2022". Le parser identifie les intentions sous-jacentes : un calcul d’agrégat (la moyenne des délais de départ), des critères exacts (les villes d’origine et de destination), et une condition temporelle (mois et année). À partir de ces informations, il construit une requête SQL complète et performante pour répondre à la demande.

## Avantages du Parser SQL

L’intégration de ce parser SQL dans Flight Catcher apporte de nombreux bénéfices. Tout d’abord, elle permet aux utilisateurs de formuler leurs requêtes dans un langage intuitif, sans nécessiter de connaissances avancées en SQL. Ensuite, les optimisations intégrées réduisent significativement les temps de traitement, offrant une expérience fluide même pour des requêtes complexes. Enfin, le système s’adapte facilement aux évolutions de la base de données grâce à son mécanisme de mapping des colonnes.

# Approfondissement et Optimisations du Parser SQL

## Grammaire et Méthode d’Analyse

Le parser repose sur une grammaire contextuelle définie par des règles claires pour analyser les requêtes en langage naturel. La méthode adoptée s’appuie sur des expressions régulières pré-compilées pour détecter des motifs spécifiques, comme les dates ou les opérateurs logiques. Cette approche garantit une analyse rapide et fiable des entrées utilisateur.

## Implantation

L’implémentation du parser est structurée en plusieurs modules interconnectés :

Module d’Analyse Lexicale : Identifie les tokens clés dans les requêtes.

Module d’Analyse Syntaxique : Détecte les structures logiques et les traduit en clauses SQL.

Module d’Optimisation : Améliore l’efficacité des requêtes par des techniques comme le caching et le regroupement.

## Limites et Perspectives

Malgré ses performances, le parser peut être amélioré pour mieux gérer les ambiguïtés dans les requêtes complexes. L’ajout de modèles d’apprentissage automatique pourrait également accroître sa précision en s’adaptant aux variations de langage naturel.

## Gains de Vitesse

Les optimisations introduites réduisent les temps de traitement selon les tests effectués sur un sous-ensemble de requêtes. Des études récentes (bibliographie 11 et 12) décrit des techniques similaires.

# scénarios d'application

Le projet Flight Catcher, grâce à l’intégration de fonctionnalités avancées telles que le parser SQL, ouvre la voie à une multitude de scénarios d’utilisation concrets. Ces cas d’application illustrent la polyvalence de l’outil et son aptitude à répondre à des besoins variés dans le domaine de l’analyse de données de vols.

## Analyse de la Ponctualité des Compagnies Aériennes

L’un des usages fréquents consiste à évaluer la ponctualité des compagnies aériennes. Un analyste peut formuler une requête naturelle comme :

"Quelle est la ponctualité moyenne des vols d’Air France sur l’année 2021 ?".

Le système identifie les critères (compagnie aérienne et année), extrait les colonnes pertinentes (compagnie et délais d’arrivée) et calcule un agrégat (moyenne). Cette analyse permet de classer les compagnies selon leur fiabilité et de fournir des informations utiles aux passagers et aux gestionnaires.

## Identification des Routes les Plus Perturbées

Dans ce scénario, les gestionnaires d’aéroport peuvent utiliser Flight Catcher pour identifier les routes les plus sujettes à des retards. Une requête comme :

"Quelles sont les dix routes avec les plus longs retards moyens au départ en 2022 ?" met en lumière les liaisons problématiques. En combinant des fonctions d’agrégation et des tris, le système fournit une liste priorisée des routes nécessitant des actions correctives.

## Analyse des Saisonnalités

Les gestionnaires de flotte aérienne peuvent analyser les variations saisonnières des vols en termes de demande ou de ponctualité. Une requête telle que :

"Quels mois enregistrent les retards les plus élevés" permet de dégager des tendances sur des plages temporelles spécifiques, en identifiant les périodes à fort risque de perturbation. Cela aide à ajuster les ressources et les plannings en conséquence.

## Suivi de la Performance d’un Aéroport

Les responsables d’aéroports peuvent utiliser l’outil pour surveiller la performance de leur infrastructure. Par exemple :

"Quel est le pourcentage de vols au départ de l’aéroport JFK ayant un retard supérieur à 15 minutes ?".

Cette analyse repose sur des comparaisons et des agrégats simples, mais elle fournit des indicateurs clés sur l’efficacité de l’aéroport et sa gestion des flux de passagers.

## Comparaison entre Compagnies sur un Segment

Les agences de voyages ou les passagers peuvent comparer les performances des compagnies sur une même liaison. Une requête telle que :

"Quelle compagnie aérienne a le moins de retards pour les vols de San Francisco à Seattle ?" aide à identifier le meilleur choix pour un trajet donné, renforçant l’expérience utilisateur.

## Simulation de Scénarios pour l’Optimisation des Ressources

En utilisant des requêtes spécifiques, les équipes d’opérations peuvent simuler différents scénarios pour allouer les ressources. Par exemple :

"Combien de retards supplémentaires sont enregistrés si le trafic augmente de 10 % pendant l’été ?".

Ces simulations permettent de mieux anticiper les besoins en personnel ou en infrastructures.

# tests de performances

## Les Configurations

Trois configurations ont été mises en place pour tester les performances du système. Chaque configuration représente une approche différente en termes de ressources allouées et de paramètres ajustés.

1. Config 1 : Basique

Cette configuration utilise des ressources minimales pour simuler un environnement avec des contraintes matérielles. Elle est adaptée aux petits jeux de données ou aux scénarios de test où l’efficacité n’est pas primordiale.

Paramètres :

Cœurs d’exécuteurs (spark.executor.cores) : 2

Cela limite la capacité de calcul simultané de chaque exécuteur.

Mémoire d’exécuteur (spark.executor.memory) : 4 Go

Une mémoire réduite peut entraîner des échanges fréquents avec le disque, réduisant les performances.

Cœurs du driver (spark.driver.cores) : 2

Une allocation minimale pour gérer la coordination des tâches.

Parallélisme par défaut (spark.default.parallelism) : 4

Faible parallélisation adaptée aux jeux de données de taille réduite.

Partitions pour le shuffle (spark.sql.shuffle.partitions) : 4

Nombre limité de partitions, ce qui peut ralentir les opérations nécessitant des regroupements ou des jointures.

2. Config 2 : Équilibrée

Cette configuration vise à trouver un compromis entre les performances et les ressources utilisées. Elle est idéale pour des jeux de données de taille moyenne et des analyses variées.

Paramètres :

Cœurs d’exécuteurs : 4

Double du précédent, offrant une meilleure capacité de calcul pour chaque exécuteur.

Mémoire d’exécuteur : 6 Go

Augmentation de la mémoire pour permettre le traitement de données plus volumineuses sans échanges fréquents avec le disque.

Cœurs du driver : 2

Identique à la configuration basique, car le rôle du driver ne nécessite pas de ressources accrues dans ce cas.

Parallélisme par défaut : 8

Double du précédent, augmentant la capacité à gérer des transformations parallèles.

Partitions pour le shuffle : 8

Davantage de partitions pour mieux répartir les charges de travail sur plusieurs exécuteurs.

3. Config 3 : Maximale

Cette configuration exploite au maximum les ressources disponibles pour tester les limites des performances. Elle est idéale pour des jeux de données volumineux ou des requêtes complexes.

Paramètres :

Cœurs d’exécuteurs : 6

Capacité maximale pour un traitement hautement parallèle.

Mémoire d’exécuteur : 8 Go

Allocation mémoire élevée pour des analyses lourdes ou des requêtes nécessitant des ressources importantes.

Cœurs du driver : 2

Inchangé, car l’augmentation des cœurs du driver n’a pas un impact significatif dans ce contexte.

Parallélisme par défaut : 12

Optimisé pour tirer parti du nombre accru de cœurs.

Partitions pour le shuffle : 12

Un nombre élevé de partitions permet de mieux répartir les charges de travail, mais peut introduire une surcharge de gestion.

## Explications des Paramètres

spark.executor.cores

Nombre de cœurs alloués à chaque exécuteur. Plus il y a de cœurs, plus un exécuteur peut traiter de tâches simultanément. Une valeur trop élevée peut entraîner une surcharge si le matériel sous-jacent ne suit pas.

spark.executor.memory

Quantité de mémoire disponible pour chaque exécuteur. Une mémoire insuffisante peut entraîner des erreurs de mémoire ou une performance dégradée.

spark.driver.cores

Nombre de cœurs alloués au processus maître (driver). Le driver coordonne les tâches et collecte les résultats. Trop de cœurs ici peuvent limiter les ressources pour les exécuteurs.

spark.default.parallelism

Nombre de tâches parallèles par défaut pour les transformations. Une parallélisation accrue peut améliorer les performances sur de grands ensembles de données, mais peut entraîner une surcharge sur des jeux de données plus petits.

spark.sql.shuffle.partitions

Nombre de partitions générées lors des opérations de shuffle (comme les agrégations et les jointures). Une valeur trop élevée augmente le coût des entrées/sorties, tandis qu’une valeur trop faible limite la parallélisation.

## Tests de Performances

Tableau des Résultats des Temps d’Exécution (en secondes)

Une image contenant texte, capture d’écran, Police, nombre

Le contenu généré par l’IA peut être incorrect.

Une image contenant texte, capture d’écran, nombre, Police

Le contenu généré par l’IA peut être incorrect.

## Analyse des Résultats

Config 1 (Basique)

La configuration avec des ressources minimales montre des temps d'exécution plus longs pour des requêtes complexes.

La faible parallélisation et la mémoire réduite limitent les performances, notamment sur des opérations demandant des agrégations complexes (comme la moyenne de distance par origine).

Config 2 (Équilibrée)

Cette configuration offre un bon équilibre entre les cœurs de calcul et la mémoire, permettant des temps d’exécution réduits dans certaines requêtes (par exemple, la moyenne de distance par origine).

Cependant, pour certaines tâches comme le retard moyen par compagnie, des temps plus longs sont observés. Cela peut être dû à des problèmes de surcharge du parallélisme ou de partitionnement.

Config 3 (Maximale)

L’augmentation des ressources permet un traitement rapide dans certaines requêtes, mais une surcharge de partitionnement et de parallélisation peut entraîner une perte de performance pour des requêtes plus simples (comme les vols vers LAX ou SFO).

Les requêtes demandant des traitements intensifs, comme le calcul des retards moyens, n’améliorent pas significativement leurs performances malgré les ressources supplémentaires.

Impact des "Executor Cores" :

Pour show average distance by origin, l'augmentation des "Executor Cores" de 2 à 4 (avec une faible mémoire de 2 Go) entraîne une dégradation des performances (temps d'exécution de 1.4036 s à 2.6090 s). Cela pourrait être dû à une surcharge de coordination entre les threads.

Dans le cas de show flights to LAX or SFO, le passage à 4 "Executor Cores" améliore les performances dans certaines configurations (temps de 0.8515 s à 0.4737 s avec 2 Go de mémoire), ce qui suggère que cette requête bénéficie davantage de la parallélisation.

Impact de la Mémoire des Exécuteurs :

L'augmentation de la mémoire (de 2 Go à 4 Go) n'a pas d'impact significatif sur show average distance by origin (temps passant de 1.4036 s à 1.4147 s avec 2 "Executor Cores"), indiquant que cette requête ne souffre pas d'une contrainte de mémoire.

Pour show flights to LAX or SFO, l'augmentation de mémoire réduit légèrement les temps d'exécution dans certaines configurations (par exemple, de 0.8515 s à 0.8351 s avec 2 "Executor Cores"), mais l'amélioration est marginale.

Combinaison Mémoire et Cœurs :

La meilleure performance pour show average distance by origin est obtenue avec 4 "Executor Cores" et 4 Go de mémoire (temps de 1.2222 s). Cela montre qu'un équilibre entre cœurs et mémoire est crucial pour cette requête.

Dans le cas de show flights to LAX or SFO, la meilleure configuration est 4 "Executor Cores" avec 2 Go de mémoire (temps de 0.4737 s). L'augmentation de mémoire à 14 Go, combinée à 4 "Executor Cores", augmente le temps d'exécution à 1.3371 s, ce qui peut être attribué à un surdimensionnement inutile des ressources.

Sensibilité aux Paramètres :

Les résultats montrent que show average distance by origin est moins sensible aux variations de mémoire, tandis que show flights to LAX or SFO est plus affectée par une augmentation des "Executor Cores".

Une allocation excessive des ressources, comme dans la configuration avec 4 "Executor Cores" et 14 Go de mémoire, peut entraîner une surcharge de gestion, augmentant les temps d'exécution pour des requêtes simples.

Analyse de la Requête : show flights with departure delay greater than 30 and arrival delay less than 15

Cette requête identifie les vols dont le retard au départ dépasse 30 minutes, mais dont le retard à l’arrivée est inférieur à 15 minutes. Elle nécessite le filtrage et la comparaison de plusieurs millions d’enregistrements dans le jeu de données. La requête implique des conditions sur deux colonnes (departure\_delay et arrival\_delay), ce qui peut engendrer un traitement important, surtout avec un grand volume de données.

Temps d'Exécution Mesuré

Durée côté serveur : 4.1364 secondes

Durée côté client : 8.4394 secondes

L'écart entre le temps serveur et client est significatif, le temps côté client étant environ deux fois plus élevé que celui du serveur.

Temps côté serveur :

Le temps côté serveur représente le traitement brut de la requête au niveau de la base de données. Cela inclut :

L’application des filtres sur les colonnes.

Le calcul des résultats correspondants.

L’organisation des données à transmettre au client.

La performance est acceptable pour un moteur de traitement analytique exécutant une opération sur des millions d’enregistrements.

Temps côté client :

Le temps côté client inclut le transfert des données traitées depuis le serveur vers le client, ainsi que leur gestion côté application.

L’écart important avec le temps côté serveur peut s’expliquer par :

Volume des données transférées :

Les résultats générés par la requête peuvent être volumineux, même après filtrage.

Délai réseau :

Si la communication entre le serveur et le client n’est pas optimisée (par exemple, en cas de faible bande passante ou de latence élevée), le transfert des résultats peut prendre plus de temps.

Traitement local :

Le client peut effectuer des opérations supplémentaires, telles que le formatage des données ou leur affichage, ce qui allonge le temps total.

Bibliographie

1. **Flight Delay Dataset 2018-2022**. Kaggle. Disponible à : <https://www.kaggle.com/datasets/robikscube/flight-delay-dataset-20182022>, accédé le 5 novembre 2024.
2. **FastAPI Framework, High Performance, Easy to Learn, Fast to Code, Ready for Production**. FastAPI Documentation. Disponible à : <https://fastapi.tiangolo.com/>, accédé le 8 novembre 2024.
3. **DuckDB: The SQLite for Analytics**. DuckDB Official Website. Disponible à : <https://duckdb.org/>, accédé le 8 novembre 2024.
4. **Apache Spark™ - Unified Analytics Engine for Big Data**. Apache Spark Official Website. Disponible à : <https://spark.apache.org/>, accédé le 24 novembre 2024.
5. **Dash: Low-Code Framework for Dashboards in Python**. Dash Plotly Documentation. Disponible à : <https://dash.plotly.com/>, accédé le 4 décembre 2024.
6. Mühleisen, H., & Raasveldt, M. (2020). DuckDB: Analytical SQL in the Local Environment. VLDB Conference Proceedings.  
   Accédé le 17 novembre 2024. <https://www.vldb.org/pvldb/vol13/p2485-raasveldt.pdf>
7. DuckDB Project Page. Why DuckDB?  
   Accédé le 22 novembre 2024. <https://duckdb.org/why_duckdb>
8. FastAPI Documentation. FastAPI: High Performance Framework.  
   Accédé le 10 novembre 2024. <https://fastapi.tiangolo.com/performance>
9. Databricks Blog. (2023). Spark vs. Local Analytical Databases.  
   Accédé le 12 décembre 2024. <https://databricks.com/blog/2023/01/spark-vs-local-analytical-databases.html>
10. Apache Spark Documentation. Apache Spark Scalability.  
    Accédé le 28 novembre 2024. <https://spark.apache.org/docs/latest/>
11. Faster and Better Grammar-based Text-to-SQL Parsing via Clause-level Parallel Decoding and Alignment Loss. Accédé le 4 janvier 2025. https://arxiv.org/pdf/2204.12186
12. PATSQL: Efficient Synthesis of SQL Queries from Example Tables with Quick Inference of Projected Columns. Accédé le 8 janvier 2025.