大容量文字データの
エージェント AI での取り扱い方法の検討

—— 列指向 DB による事前索引化と全文検索性能の実測 ——

永山 和樹　　大和田 昭邦
株式会社 DynaxT 新技術商品開発部　　株式会社 DynaxT

2026 年 7 月 27 日

概要

目的：現状の業務では、すでに大量のデータが蓄積されている。しかしそれが綺麗な階層構造に整理されていることは少ない。多くの場合、フラットなフォルダに大量のファイルが置かれていたり、1 つの巨大なファイルやシステムに閉じ込められていたり、ファイル名が内容を表していなかったりする。こうしたデータは業務上きわめて有益であるにもかかわらず、エージェント型 AI でうまく扱えず、活用が制限されてきた。**本研究の大目的は、この「整理されていないまま蓄積された大容量データ」をエージェント型 AI が実用的な速度で扱えるようにし、埋もれている情報を活用可能にすることである。**

本稿の内容：フォルダ階層が適切に設計されていればエージェントは必要箇所のみを読めばよいが、上記のとおり実運用では「巨大な単一ファイル」や「フラットに置かれた大量ファイル」が与えられ、素朴な全文検索（grep 相当）では実用的な応答時間に収まらない。本研究では、**組込み型の列指向データベース（OLAP）**と列指向ファイル形式 **Parquet** を用いて事前に索引化する方式を採り、**10GB / 100GB / 300GB** の 3 規模・2 系統のデータセット（Synthea 生成の関係表 CSV、および日本語を含む独自生成 JSON）に対して、**Parquet 化に要する時間**と**全文検索に要する時間**を実測した。

要点：結論を先に述べると、**DB 化しなければ、そもそも検索が成立しなかった**。要点を表 1 に示す。

表 1　DB 化の有無による検索可否の違い

状態	対象データ	全文検索 1 回の所要時間
DB 化なし （直読み）	12 GB 1.8 万ファイル	高性能モデル：約 9 分（対話不能） エッジ AI：**検索できない**（ハング・完了せず）
DB 化あり （全文走査）	**300 GB** 10 億行	**約 2.7 分**（データ量 25 倍にして、なお高速）
DB 化あり （列を限定）	**300 GB** 10 億行	**0.006〜0.411 秒**（対話的に利用可能）

「DB 化なし」は実運用で観測された挙動、「DB 化あり」は本実験の実測値である。

すなわち出発点として、DB 化しない素朴なファイル走査は**約 12GB・1.8 万ファイルの時点ですでに破綻**しており、高性能モデルでも全文検索 2 回に約 18 分、エッジ AI に至っては**ハングして完了しなかった**。この規模で破綻する以上、想定される 100〜300GB・数億行の CSV ではより深刻になる。

これに対し、原データをあらかじめ Parquet 形式に索引化し、列指向 DB から検索したところ、その **25 倍にあたる 300GB でも全文検索が約 2.7 分**で完了した。さらにスキーマを利用し

て**検索対象列を限定**すると、同じ 300GB・10 億行に対する条件検索は **0.006 秒**、テーブル結合を含む集約でも **0.5 秒未満**と、全文走査比で最大約 7×10^5 倍の高速化が得られた。Parquet 化自体は 300GB で約 0.8〜2.9 時間を要するが一度きりの前処理であり、生成物は原データの約 1/10〜1/13 に圧縮される。本稿は、この定量結果に基づき「エージェント AI には全文走査を最終手段として残しつつ、スキーマを与えて列指定 SQL を書かせる」運用方針を提案する。

目次

1　経緯

大容量のテキストデータをエージェント型 AI で扱いたいという要望がある。総データ量は 100〜300GB 程度を想定している。

大容量であっても、適切にフォルダ分けがされていれば、エージェント AI はフォルダ階層をたどって必要な情報のみにアクセスするため、特別な対処なしに対応可能である。しかし今回想定されるのは、**(a) 1 ファイルで巨大なテキストデータ**、または **(b) フラット（階層構造なし）に置かれたテキストデータ**である。人手で適切な階層を与えることも難しい。

今回、実例のデータセットとして医薬品医療機器総合機構（PMDA）[4] の添付文書データ（約 12GB）を示す。これをエージェント AI で活用する場合に問題が生じている。このデータセットはフラットな階層であり、単一フォルダに約 1.8 万個の PDF が置かれている。API で提供されている最先端の高性能モデルと高速ディスクを備えた PC であれば、分割読み込みなどにより対応可能ではあるが、動作は遅く **1.8 万件のファイル内を 2 回全文検索するだけで 18 分程度**を要することがあった。さらに、オープンモデルの AI では全件を一度に読み込もうとして、ファイル数の多さにハングする事例も観測された。

これを解決するため、**事前に DB 化しておく**方針を採ることとした。通常の SQL サーバとは利用用途が異なるため、以下の条件を満たす技術を探索した。

(1) 事前にファイルを取り込み、全文検索を高速に行えること
(2) エージェント型 AI、および AI が多用する Python との相性が良いこと
(3) ファイルサイズがなるべく小さくなること
(4) SQL の形式でクエリが行えること

その結果、**組込み型の列指向 DB** が候補として見つかったため、本稿ではその検証を行う。

2　関連研究：なぜこのアプローチが有効と考えられるか

本稿が採る「**事前に構造化して索引化し、エージェントには SQL を書かせる**」という方針は、本件のために思いついたものではなく、近年の知見と整合している。考えうる代替案を順に検討し、既存研究に照らして本アプローチを位置づける。

2.1　代替案 1：長いコンテキストにまとめて読ませる

最も素朴な方法は、ファイル内容をそのままエージェントのコンテキストに積むことである。しかしこれは §1 のとおり実運用で破綻した。文献上も裏づけがある。Liu ら [8] は、入力コンテキストが長くなるほど言語モデルの性能が**位置に依存して劣化**し、とくに**中央に置かれた情報の利用が著しく落ちる**ことを示した。すなわち、大量のファイルを機械的に流し込む方式は、コンテキスト長の上限に達する前から**精度の面で**割に合わない。物理的な限界（本件で観測したハング）と精度的な限界の双方から、**読ませる量を絞り込む機構が先に必要**である。

2.2 代替案 2：ベクトル検索（RAG）で絞り込む

絞り込みの定番はベクトル検索 RAG[9] である。埋め込みの近さで関連文書を引き、それだけをコンテキストに与える方式で、意味的に曖昧な問い（「〜に似た事例」）に強い。

しかし本件の想定業務では、次の性質をもつ問い合わせが中心となる。

- **もれなく全件**：「該当するものを**すべて**挙げよ」。1 件の取りこぼしも許されず、上位 k 件では答えにならない。
- **厳密一致**：特定の識別子・コード・日付との完全一致。
- **集計**：件数、合計、期間別の分布。

いずれも近似最近傍探索が得意とする形ではなく、**正確な条件判定と集計**、すなわち SQL の領域である。本稿の方式と RAG は排他ではなく、曖昧な問いには RAG、全件列挙・厳密一致・集計には SQL という併用が自然である。

2.3 代替案 3 の検討：エージェントに SQL を書かせられるのか

本稿の提案（§7.3）は、エージェントにスキーマを与えて SQL を生成させる形を採る。この実現可能性は、自然言語から SQL を生成する text-to-SQL の研究蓄積によって支えられている。Yu ら [10] の Spider は 200 データベース・138 ドメインにわたる横断的なベンチマークを整備し、この課題を標準的な評価対象に押し上げた。さらに Li ら [11] の BIRD は、**95 データベース・総計 33.4 GB** という実データに近い規模で評価を行っており、本件の関心（大規模・実データ）に近い。

BIRD の報告は本稿にとって二面的である。一方で、LLM が実規模の DB に対して SQL を生成し実行できること自体は成立している。他方で、実行精度は **GPT-4 で 54.89%、人間の 92.96% には遠く及ばない**とされ、とくに**スキーマの理解とデータ値の把握**が誤りの主要因として挙げられている。

ここから 2 つの示唆が得られる。

(1) **アプローチとしては成立する**。エージェントに SQL を書かせる形は研究上も実用上も想定されたやり方であり、本稿が測るべきは「その先の実行速度が業務に耐えるか」である。

(2) **スキーマの提示設計が精度を決める**。したがって、§7.3 で述べる運用指針のうち「スキーマを与える」は単なる便宜ではなく、**精度を左右する本質的な要件**である。したがって実運用に移す際は、生成 SQL の正答率をあわせて評価する必要がある。

2.4 高速化の根拠

最後に、本稿が観測する高速化そのものは、列指向データベースの既知の性質に帰着する。Abadi ら [6] は、列単位の格納・圧縮・遅延マテリアライズなどにより、**参照する列だけを読む**走査が分析系ワークロードで大幅に有利になることを整理している。本稿で用いる列指向 DB はこの設計を組込み型として実装したものであり、Parquet[1] は同じ列指向の考え方をファイル形式として標準化したものである。本稿 §6.5 で得られる「列を絞れば 4〜5 桁速い」という結果は、この性質が 300GB 規

模でも素直に効くことの実測的な確認にあたる。

3　技術詳細

3.1　列指向 DB

本稿で用いるのは、オープンソースの組込み型・分析向け（OLAP）リレーショナルデータベースである。一般的な RDBMS が行単位でデータを扱う「行指向型」であるのに対し、本 DB は列単位でデータを保持する「列指向型」[6] を採用している。この構造上、特定の 1 行を頻繁に追加・更新・削除するといった細かい処理には不向きだが、**特定の列を対象とした大量データの集計や検索においては極めて高速**なパフォーマンスを発揮する。

データの保持形式には、汎用の列指向ファイル形式である **Parquet**[1] を用いた。Parquet は Apache Software Foundation が仕様を公開しており、Python の Pandas など多くのデータ分析ツールから直接読める。想定用途では取り込み後の更新・書き込みを行わず、またデータサイズをなるべく削減したいことから、本検証ではこの形式に絞って計測を行う。圧縮コーデックには `zstd`[2, 7] を用いた。

3.2　全体フロー

本検証の処理フローを図 1 に示す。原データ群を一度だけ走査して Parquet へ変換し（§5 の指標 n）、以後の検索は Parquet に対して行う（同じく指標 m）。

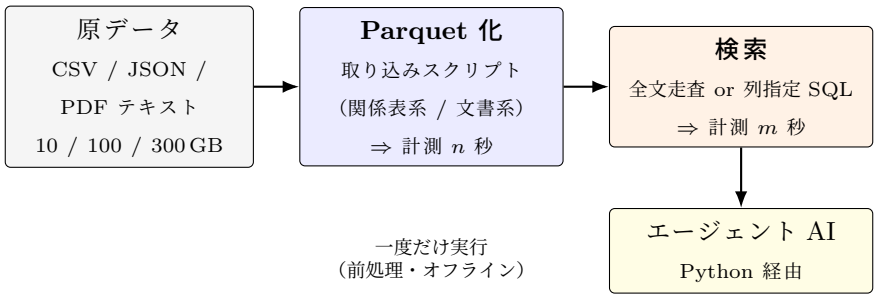

図 1　検証の全体フロー。Parquet 化は一度きりの前処理であり、以後のエージェントからの問い合わせはすべて Parquet に対して行う。

4　環境

実験に用いたハードウェア・ソフトウェア構成を表 2 に示す。すべての計測を同一マシン・同一ストレージ（NVMe SSD）上で実施した。

なお、Core i7-12650H は 10 コア 16 スレッドであるが、計測条件を揃えるため DB の実行スレッド数は全計測を通して 8 に固定した。

表 2　実験環境

ハードウェア		ソフトウェア	
CPU	Core i7-12650H	Python	3.14.5
メモリ	64 GB DDR4	列指向 DB	組込み型・ライブラリ版
SSD	NVMe 1 TB	実行スレッド数	8
		圧縮	zstd

5　検証方法

5.1　手順

1. 3 系統・計 6 組のデータセットを用意する（§5.2）。
2. それぞれについて、Parquet 化に要する時間 n を計測する。
3. Parquet 化したものに検索 SQL を実行し、実行時間 m を計測する。

5.2　データセット

用意したデータセットを表 3 に示す。

表 3　検証に用いたデータセット

#	データ種類	10 GB	100 GB	300 GB	備考
1	Synthea[5, 3]（合成医療データジェネレータ）で生成した CSV データ	33,596,294 行	344,467,303 行	1,016,063,616 行	アメリカの医療データに準拠しているため、日本とは特性が違う可能性あり
2	独自の医療想定 JSON 形式（`gen_data.py` を作成して出力）	1,280 ファイル 5,212,856,536 文字	12,800 ファイル 52,128,776,187 文字	38,399 ファイル 156,382,052,162 文字	1 に日本語データが含まれていなかったため、独自の JSON を生成する Python を作成した

「10GB / 100GB / 300GB」は生成時の目標値であり、実際に生成された原データ量はデータセット 1 が 9.91 GB / 100.03GB / 300.77 GB、データセット 2 が 10.02 GB / 100.21 GB / 300.63 GB であった。いずれも目標値との差は 1% 未満のため、以降の表では目標値の表記を用いる。

データセット 1 は**関係表（18 テーブル）**であり、1CSV=1 テーブルとして Parquet 化した。データセット 2 は**文書指向**であり、1 ファイル=1 行の `documents(content)` という単一テーブルに全本文を格納した。両者は「大容量テキスト」という括りは同じだが、後述するとおり検索コストの支配要因が異なる。

表 4　計測指標の定義

指標	定義	測り方
n：Parquet 化時間	原データ群の走査、全内容の取り込み、Parquet 書き出し完了まで	スクリプト内 `time.time()` による経過秒（`build_seconds`）
m：全文検索時間	全テーブル・全列を対象とした部分一致（AND）の全走査	3 回実行の**中央値**（`median_s`）
m'：列指定検索時間	スキーマを利用し対象列を限定した SQL	3 回実行の**中央値**

5.3　計測指標の定義

6　結果

本節は次の順で述べる。まず **DB 化しない場合に何が起きていたか**を確認し（§6.1）、そこから**規模を上げた場合の見通し**を述べる（§6.2）。そのうえで、DB 化した場合の実測値を Parquet 化時間（§6.3）、全文検索時間（§6.4）、列指定 SQL（§6.5）の順に示し、最後に三段階を並べて要約する（§6.7）。

6.1　出発点：DB 化しなければ 12GB の時点で破綻していた

まず確認すべきは、**DB 化しない素朴なファイル走査は、300GB どころか 12GB の時点ですでに破綻していた**という事実である。§1 で述べた PMDA の事例（約 12GB・約 1.8 万ファイル・フラット階層）で観測された挙動を表 5 に示す。

表 5　DB 化前（ファイル直読み）の挙動。PMDA データ 約 12 GB・約 1.8 万ファイル

エージェント	挙動	所要時間
API 提供高性能モデル	分割読み込みにより一応は完遂できる	全文検索 2 回で**約 18 分**
オープンモデル	全件を一度に読み込もうとし、ファイル数の多さに**フリーズ**	**完了しない**

本表は実運用で観測された挙動であり、本実験の管理された計測値ではない。高性能ディスクを備えた PC での観測である。

つまり、高性能モデルであっても**検索 1 回あたり約 9 分**かかり対話的な用途に耐えず、エッジ AI に至っては**そもそも完了しない**。12GB という、本実験で扱う最小規模（10GB）とほぼ同じサイズで、すでにこの状態であった。

6.2　規模を上げればさらに悪化する

12GB・1.8 万ファイルで破綻するのであれば、本件で想定される **100〜300GB・数億行の CSV** ではより深刻になると考えるのが自然である。根拠は次の 3 点である。

(1) **データ量が 25 倍**。PMDA の 12GB に対し 300GB は 25 倍であり、ファイル走査の時間はデータ量にほぼ比例する。

(2) **レコード数が桁違い**。本実験のデータセット 1 は 300GB で**約 10 億行**（表 3）であり、PMDA の 1.8 万ファイルとは 4〜5 桁の開きがある。

(3) **ハングの原因が悪化する**。エッジ AI のハングはファイル数・トークン数に起因するため、規模が上がれば解消せずむしろ確実に起きる。

すなわち、**DB 化なしという選択肢は最初から存在しない**。

6.3 Parquet 化時間

結果を表 6 に示す。

表 6　Parquet 化に要した時間（秒）

#	データ種類	10 GB	100 GB	300 GB
1	Synthea CSV データ	467.80	3,317.12	10,327.51
2	独自の医療想定 JSON 形式	84.22	1,648.62	2,722.69
3	PMDA PDF（文字起こし済み）	○	—	—
（参考）1 の実効スループット		21.9 MB/s	30.8 MB/s	29.8 MB/s
（参考）2 の実効スループット		121.6 MB/s	62.1 MB/s	112.7 MB/s

実効スループットは公称サイズを所要秒で除した参考値である。

時間換算では、300GB の Parquet 化に データセット 1 で**約 2 時間 52 分**、データセット 2 で**約 45 分**を要した。100GB ではそれぞれ約 55 分・約 27 分である。

6.4 全文検索時間

全テーブル・全列を対象とした全文走査の結果を表 7 に示す。

表 7　全文検索（全列全走査）に要した時間。3 回実行の中央値［秒］、括弧内はヒット件数

#	クエリ	10 GB	100 GB	300 GB
1	**高頻度語**	18.983 (1,790,132)	546.151 (18,397,496)	4,339.181† (54,247,730)
	低頻度語	18.385 (88,107)	618.869 (912,007)	—
	高頻度語・中頻度語の `AND`	21.401 (82,524)	527.908 (855,741)	—
2	**高頻度語**	5.994 (1,280)	52.836 (12,800)	163.704 (38,399)
	低頻度語	5.963 (1,280)	50.514 (12,800)	163.070 (38,399)
	高頻度語・低頻度語の `AND`	5.887 (1,280)	49.037 (12,800)	159.386 (38,399)

図 2 に、データ量に対する全文検索時間の推移を両対数で示す。

6.5 列指定 SQL（スキーマを利用した検索）

データセット 1 の Parquet に対し、対象列を限定した 5 種類の SQL を実行した結果を表 8 に示す。

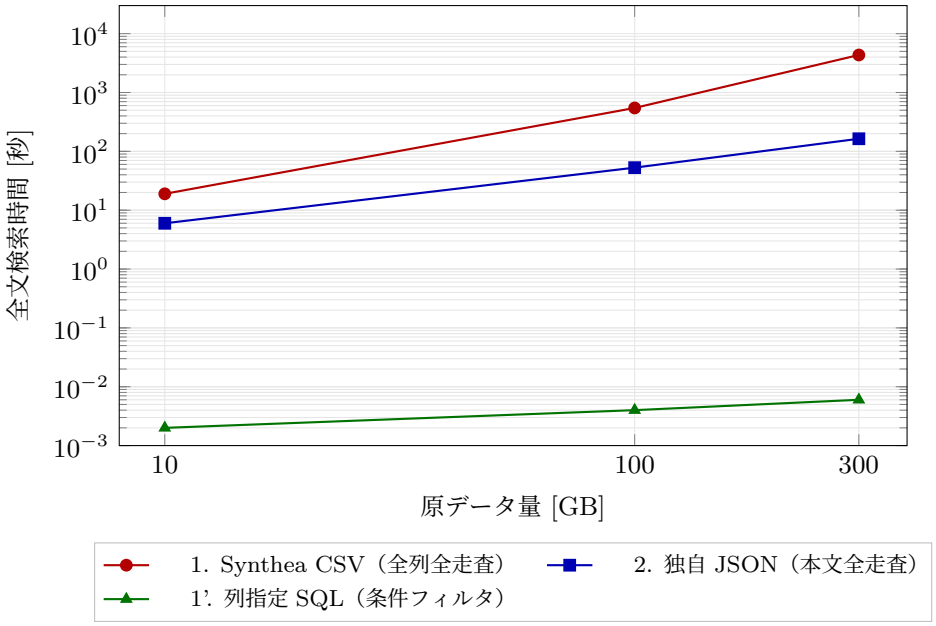

図 2　データ量に対する検索時間のスケーリング（両対数）。両対数上で傾き 1 が線形を意味する。データセット 2 はほぼ傾き 1（線形）であるのに対し、データセット 1 はそれより急な傾き（超線形）で悪化する。列指定 SQL は 3〜5 桁下方に位置し、ほぼ一定である。

表 8　列指定 SQL の実行時間。3 回実行の中央値［秒］

クエリ種別	10 GB	100 GB	300 GB
条件フィルタ	0.002	0.004	0.006
集約 (`GROUP BY`)	0.005	0.013	0.017
総件数	0.004	0.018	0.045
結合：	0.009	0.041	0.070
数値集約：平均	0.028	0.215	0.411
（参考）同データの全文走査	18.983	546.151	4,339.181

300GB・10 億行のデータに対して、条件フィルタは **0.006 秒**、5 種のうち最も重い数値集約でも **0.411 秒**で完了した。全文走査（4,339 秒）との比は、それぞれ**約** 7.2×10^5 **倍・約** 1.1×10^4 **倍**である。5 種すべてが **1 秒未満**であり、対話的な応答として十分な速度である。

6.6　生成されたファイルサイズ

Parquet 化後のサイズを表 9 に示す。

表 9　Parquet 化後のファイルサイズ［GB］と圧縮率

#	データ種類	10 GB	100 GB	300 GB	圧縮率 (300GB)
1	Synthea CSV データ	0.745	7.758	22.884	約 1/13
2	独自の医療想定 JSON 形式	1.032	10.323	30.967	約 1/10

いずれも原データの**約 1/10** に収まっており、§1 の条件 (3)（ファイルサイズがなるべく小さくな

ること）を満たす。300GB の原データが約 23〜31GB の Parquet になるため、原データを削除できる運用であれば保管コストはむしろ下がる。

さらに、**原データを削除できない運用であってもメリットがある**。一般に「検索用の DB を別に持つ」となると保管容量が 2 倍になることを懸念しがちだが、本方式で追加が必要なのは原データの**約 1 割**にとどまる。300GB の原データに対して追加は約 23〜31GB であり、合計でも約 1.1 倍である。すなわち **2 倍の容量を用意する必要はなく**、原データを残したまま導入できる。

6.7　三段階の比較

以上を 1 表にまとめると表 10 になる。

表 10　DB 化前・DB 化後（全文走査）・DB 化後（列指定）の比較

段階	対象データ	検索 1 回の所要時間	実用性
① DB 化なし（ファイル直読み）	12 GB / 1.8 万ファイル	約 9 分（高性能モデル） **完了せず**（エッジ AI）	不可
② DB 化＋全文走査	**300 GB** / 10 億行	2.7 分（文書指向） 72 分（関係表を疑似全文化）	条件付きで可
③ DB 化＋列指定 SQL	**300 GB** / 10 億行	**0.006〜0.411 秒**	対話的に可

①は実運用での観測（本実験の管理された計測値ではない）。②③は本実験の実測値。①と②③でデータ量が 25 倍異なる点に注意されたい。

読み取るべき点は次の 3 つである。

(1) **①→②で、25 倍のデータを扱いながら速くなっている**。12GB で 9 分かかっていた検索が、300GB で 2.7 分になった。文書指向に寄せた場合の単位データ量あたりでは**約 80 倍**の改善である。

(2) **②→③で、さらに 4〜5 桁速くなる**。スキーマを使って列を絞れば 1 秒を切り、**対話的な応答**が成立する。

(3) **①の「完了しない」が消える**。エッジ AI にとっても、1.8 万ファイルを開く作業が SQL1 本の発行に置き換わる。

結果の要約

DB 化前は **12GB の時点でフリーズしていた**。DB 化後は、その 25 倍にあたる **300GB でも全文検索が数分**で完了し、**項目を絞れば 1 秒以下**で返るようになった。

7　考察

7.1　Parquet 化は「一度きり」のコストである

300GB で最大約 2.9 時間という数値は、対話的な処理としては許容できない。しかし Parquet 化はデータ連携のたびにバッチで実行すればよい前処理であり、エージェント AI の応答時間には含まれない。差分連携が可能であれば、増分ファイルのみを追加 Parquet として書き出すことで、実運用の所要時間はさらに短縮できる。

データセット 1（CSV）がデータセット 2（JSON）より一貫して低速（21.9〜30.8 MB/s 対 62.1〜121.6 MB/s）であるのは、CSV パーサが行ごとに 18 テーブル分の型推論・型変換を行うのに対し、JSON 側は本文をそのまま 1 つの文字列列に格納するだけで済むためと考えられる。すなわち**Parquet 化コストは、バイト数よりもレコード数と列数に強く依存する**。データセット 2 の 100GB のみスループットが落ちている（62.1 MB/s）が、これは他プロセスの影響を含む単発計測であり、本質的な傾向ではないと判断している。

7.2　スキーマを与えれば桁が変わる

最も重要な結果は表 8 である。同じ 300GB・10 億行の Parquet に対し、検索対象列を限定するだけで、全文走査の 4,339 秒が **0.006 秒**になった。列指向フォーマットの本領であり、Parquet は列単位に格納・圧縮されているため、参照しない列は**ディスクから読まれもしない**。さらに min/max 統計による row-group の枝刈りが効く。

したがって、性能を決めるのは「どの DB 製品を使うか」ではなく「**スキーマを設計して検索対象列を限定できるかどうか**」である。本実験の全文走査の数値は、あくまで**スキーマをまったく検討しなかった場合の上限値（最悪ケース）**として読むべきである。

7.3　エージェント AI の運用への含意

以上から、エージェント AI に大容量テキストを扱わせる際の設計指針を次のように提案する。

1. **原データは事前に Parquet 化しておく**（バッチ・オフライン）。サイズは約 1/10 になり、原データを直接読ませる必要がなくなる。
2. **エージェントにはスキーマ（テーブル定義・列の意味・代表値）をコンテキストとして与える**。これによりエージェントは列指定 SQL を書けるようになり、応答は 0.01 秒オーダに収まる。なお §2 で述べたとおり、スキーマとデータ値の提示は text-to-SQL の精度を左右する主要因 [11] であり、速度のためだけでなく**正しい SQL を書かせるため**にも必要である。
3. **全文走査は最終手段として残す**。どの列にあるか不明な語を探す場合のみ使い、事前に「数分〜数十分かかる」ことをエージェント側に知らせて安易に選択させない。
4. **フラットな大量ファイルは文書指向テーブルに寄せる**。1 ファイル=1 行の `documents(path, content)` 形式であれば、全文走査でも線形（本実験で 1.8 GB/s 相当）に収まり、ファイル数に起因するハングも起きない。

§1 で述べた「1.8 万件の PDF を 2 回全文検索して 18 分」という PMDA の事例は、この方式であれば Parquet1 ファイルに対する数秒〜十数秒の走査に置き換わる。「ファイルを列挙して読む」から「SQL を 1 本発行する」へ操作が単純化されるため、小型オープンモデルでも扱える見込みが高い。

8 結論

エージェント型 AI に 100〜300GB 規模のテキストデータを扱わせる方法として、列指向 DB と Parquet による事前索引化を検証した。得られた知見は以下のとおりである。

1. DB 化しない素朴なファイル走査は、**約 12GB・1.8 万ファイルの時点ですでに破綻**していた（高性能モデルで検索 1 回あたり約 9 分、エッジ AI はハングして完了せず）。想定される 100〜300GB・数億行ではさらに悪化するため、DB 化は選択肢ではなく前提である。
2. Parquet 化には 300GB で約 45 分〜2 時間 52 分を要するが、これは一度きりのオフライン前処理であり、成果物は原データの**約 1/10〜1/13** に圧縮される。
3. DB 化すると、**12GB の 25 倍にあたる 300GB** に対する全文検索が**約 2.7 分**（文書指向）で完了した。関係表を疑似全文化した場合でも約 72 分である。検索時間はヒット件数に依存せずスキャン量で決まり、文書指向データではデータ量に対してほぼ線形、関係表を疑似全文化した場合は超線形に悪化する。
4. スキーマを利用して**検索対象列を限定**すれば、同じ 300GB・10 億行に対する条件検索は **0.006 秒**、結合を含む集約でも 0.5 秒未満で完了する。全文走査比で最大**約** 7×10^5 **倍**の高速化である。

結論として、列指向 DB と Parquet の組み合わせは、大容量テキストをエージェント AI に扱わせる基盤として**有用**である。ただしその効果は自動的には得られず、**スキーマを設計してエージェントに提示し、列指定 SQL を書かせる**という運用設計が伴って初めて実現する。全文走査は「どの列にあるか分からない語」への最終手段として位置づけるのが適切である。

参考文献

[1] The Apache Software Foundation, “Apache Parquet,” https://parquet.apache.org/

[2] Facebook, “Zstandard—Real-time data compression algorithm,” https://facebook.github.io/zstd/

[3] The MITRE Corporation, “Synthea—Synthetic Patient Population Simulator,” https://synthetichealth.github.io/synthea/

[4] 独立行政法人 医薬品医療機器総合機構（PMDA），https://www.pmda.go.jp/

[5] J. Walonoski, M. Kramer, J. Nichols, A. Quina, C. Moesel, D. Hall, C. Duffett, K. Dube, T. Gallagher, and S. McLachlan, “Synthea: An approach, method, and software mechanism for generating synthetic patients and the synthetic electronic health care record,” *Journal of the American Medical Informatics Association*, Vol. 25, No. 3, pp. 230–238, 2018. DOI: 10.1093/jamia/ocx079

[6] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, “The Design and Implementation of Modern Column-Oriented Database Systems,” *Foundations and Trends in*

Databases, Vol. 5, No. 3, pp. 197–280, 2013. DOI: 10.1561/1900000024

[7] Y. Collet and M. Kucherawy (Eds.), "Zstandard Compression and the 'application/zstd' Media Type," RFC 8878, IETF, February 2021. https://www.rfc-editor.org/rfc/rfc8878

[8] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the Middle: How Language Models Use Long Contexts," *Transactions of the Association for Computational Linguistics*, Vol. 12, pp. 157–173, 2024. DOI: 10.1162/tacl_a_00638

[9] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *Advances in Neural Information Processing Systems*, Vol. 33, pp. 9459–9474, 2020.

[10] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. Radev, "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task," *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 3911–3921, 2018.

[11] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Cao, R. Geng, N. Huo, X. Zhou, C. Ma, G. Li, K. C. C. Chang, F. Huang, R. Cheng, and Y. Li, "Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs," *Advances in Neural Information Processing Systems 36 (NeurIPS 2023), Datasets and Benchmarks Track*, 2023.